Studies in Computational Intelligence 806

Chung Yik Cho
Rong Kun Jason Tan
John A. Leong
Amandeep S. Sidhu

Large Scale Data Analytics

Springer

Studies in Computational Intelligence

Data, Semantics and Cloud Computing

Volume 806

Series editor

Amandeep S. Sidhu, Biological Mapping Research Institute, Perth, WA, Australia
e-mail: dscc@biomap.org

More information about this series at http://www.springer.com/series/11756

Chung Yik Cho · Rong Kun Jason Tan
John A. Leong · Amandeep S. Sidhu

Large Scale Data Analytics

Springer

Chung Yik Cho
Curtin Malaysia Research Institute
Curtin University
Miri, Sarawak, Malaysia

John A. Leong
Curtin Malaysia Research Institute
Curtin University
Miri, Sarawak, Malaysia

Rong Kun Jason Tan
Curtin Malaysia Research Institute
Curtin University
Miri, Sarawak, Malaysia

Amandeep S. Sidhu
Biological Mapping Research Institute
Perth, WA, Australia

ISSN 1860-949X ISSN 1860-9503 (electronic)
Studies in Computational Intelligence
ISSN 2524-6593 ISSN 2524-6607 (electronic)
Data, Semantics and Cloud Computing
ISBN 978-3-030-03891-5 ISBN 978-3-030-03892-2 (eBook)
https://doi.org/10.1007/978-3-030-03892-2

Library of Congress Control Number: 2018960754

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Contents

List of Figures

List of Tables

Chapter 1
Introduction

1.1 Project Overview

In this modern technological age, data is growing larger and faster compared to previous decades. The existing methods used to process and analyse the overflowing amount of data are no longer sufficient. The term large scale data first surfaced in the magazine "Visually Exploring Gigabyte Datasets in Real Time" [1] published in Association for Computing Machinery (ACM) in 1999. It was mentioned having large scale data without a proper methodology to analyse data is a huge challenge and a sad occasion at the same time. In the year 2000, Peter Lyman and Hal Varian [2] from University of California at Berkeley (both currently resides in Google as chief economist) attempted to measure the available data volume and data growth rate. Both senior researchers concluded that 1.5 billion gigabytes of storage was required to contain the data from film, optical, magnetic and print material annually.

Starting from 2001 onwards, large scale data was defined as data that contains high volume, high velocity and high variety. This definition was defined by Douglas Laney, an industry analyst currently working with Gartner [3]. The definition of high volume in large scale data refers to the continuous growth of data that consisted of terabytes or petabytes of information [4]. For instance, the data produced by existing social networking sites are counted in terabytes per day [5]. High velocity refers to the speed of data flow from different data sources [4]. For example, if data is constantly flowing in from a sensor to a database storage, the amount of data flow is large and fast at the same time [5]. High variety data does not mainly consist of traditional data, it also contains structured, semi-structured, unstructured or raw data. These data come from miscellaneous sites such as web pages, e-mails, sensor devices, social media sites and others, for example Facebook, Twitter, Outlook and Instagram in our modern society [5].

Two other additional elements are required to be taken into consideration when it comes to large scale data, variability and complexity. Variability takes the

C. Y. Cho et al., *Large Scale Data Analytics*,
Studies in Computational Intelligence 806,
https://doi.org/10.1007/978-3-030-03892-2_1

inconsistency of data flow into consideration as data loads are getting harder to manage [5]. Due to increasing usage of social media, for instance, Facebook generates over 40 petabytes of data daily, there are increasingly high peaks in data loads to databases [6]. As for complexity, data from various sources are very difficult to be related, matched, cleaned and transformed across systems. It is very important that the data is associated with its relevant relationships, hierarchies and data linkages otherwise they will not be sorted accordingly [5].

Large scale data has been growing ever since and it is difficult to contain such vast information. To make use of large scale data, it is required to have a proper methodology to retrieve and analyse these data. In this chapter, we are going to discuss about the research background, problem statements and the objectives of this research.

1.2 Research Background

Faced with the enormous amount of data, the traditional data analytic methodologies are no longer sufficient [7]. In this modern technological phase, data can be processed via the statistical algorithms method through dumping data into the largest high-performance computing clusters to obtain results [7]. The processed data is then stored in different data sources and they come in useful in scientific applications and business usage such as biosciences, market sales and different fields [8].

Term analytics is defined as a method of data transformation for better decision making whereas large scale data analytics is defined as a process that extracts large amounts of information from complex datasets consisting of structured, semi structured, unstructured and raw data [8]. The usage of large scale data analytics can be applicable to various fields, such as improving marketing strategies by analysing real consumer behaviour instead of predicting the needs of their customer and making gut-based decisions [9]. Information extracted from data sources through data analytics can perform and improve strategic decisions of business leaders by just adding a feature to study telemetry and the usage of user data on multiple platforms be it on mobile applications, websites or desktop applications [10]. Retrieved data can be used for recommendation engines, for example, ‘think Netflix’ and YouTube video suggestions. Large scale analytics uses intensive data mining algorithms to produce accurate results and high performance processors are required for the process [8]. Since large scale data analytics applications requires huge amount of computational power and data storage, infrastructures offered by cloud computing can be used as a potent platform [8].

Ontology has been used for large scale analytics to utilize shared vocabulary for data mapping. The word ontology originates from a philosophical term which refers to ‘the object of existence’ and from the perspective of the computer science community, it is known as ‘specification of conceptualization’ for information sharing in artificial intelligence [11]. There is a conceptual framework which is presented using

ontologies to show the significance of structured image through common vocabulary in a provided biological or medical domain. This information can be used by automated software agents and users in the domain [11]. The concepts, its relationships, the definitions of its relationships and the prospect of ontology rules and axiom definitions are included by the shared vocabulary to define the mechanism that is used to control the substances which are introduced into the ontology and the application of the substance based on logical inference [12].

For multiple fields, there are a lot of organizations that tend to maintain their data in a proprietary database. When the data in databases are available for other people to reference, the obtained data tends to be in different schemas and structures. Moreover, it is difficult to translate and integrate biomedical data as it is constantly updated and covers enormous amounts of data in the field of genomic information that contains data from genome sequencing and gene expression sequencing. Hence, the greatest challenge in this research is to ensure that any data search or querying would comprehensively cover all available databases without the need for data integration and data translation.

1.3 Problem Statement

Existing query methodologies focuses more on data integration. These methodologies can be used if the size of the targeted data sources is not large and the unified database is continually updated. For biomedical data integration, it involves genomics and proteomics data with relation to data semantics. Data semantics consists of value or meaning of data and the difference of semantics in multiple sources. Hence, the differences in concept identification, concept overloading and data transformation issues are important and requires addressing for existing data integration query methodologies [13]. There are two elements for concept identification: data identification when data from different sources are referring to the very same object and information integration conflicts found in these different sources [13]. The identification of an abstract concept identified in every single data source needs to be performed first to address these issues. The information conflict can be effortlessly solved after the shared concepts have been defined [13]. For instance, two different values are defined in two different sources to represent one attribute, which theoretically should be the same. The answer to a query, when added to the reconciliation process used by genomics, may not be correct. These accrued errors cause it to be one of the flaws with genomics as the possible differences between the two sources makes reconciling the data difficult and it needs to be stored in an integrated view [13]. This approach makes the seemly simple query into a much more complicated endeavour than it first appears to be.

Furthermore, the usage of existing query methodology is not efficient and cost effective. The existing methodology presented by researchers requires huge amounts of computing resources and time to complete data translation, data mapping and query processing. However, with the proposed query framework, process

of data querying and data management are easier compared to its predecessors. The query framework built using Language Integrated Query needs to be easily deployable on a cloud computing environment while ensuring the performance in handling and querying large scale data sources can be done smoothly.

1.4 Objective

The objective of this MPhil is to design a framework using Language Integrated Query to manage large scale data sources and implement it on a Cloud Computing environment, Microsoft Azure. This designed large scale data analytics framework can overcome the problems of other existing frameworks by being able to manage different type of large scale data sources without having structure conflict issues. The result of having the framework should be:

1. Easier to manage large scale data sources: Managing large scale data sources is no longer time consuming as the framework built using Language Integrated Query can manage large data sources all together instead of perusing data from multiple existing frameworks querying different types of data sources.
2. Easier access to the framework using web applications: A web application deployed on the Cloud Computing environment, Microsoft Azure can easily access the implemented framework to query different large scale data sources.
3. Higher processing power to operate the framework: Implementing the framework on a Cloud Computing environment, Microsoft Azure allows the framework to fully utilize the available scalable resources of Microsoft Azure to process tasks efficiently.

1.5 Outline

Six main chapters will be discussed in Table 1.1.

Table 1.1 Outline

Chapter 1	This chapter is mainly project introduction which includes project overview, research background, problem statement and objective
Chapter 2	This chapter focuses on project background and literature reviews related to project
Chapter 3	This chapter introduces the proposed methodology applied for this project
Chapter 4	This chapter introduces the proposed query framework in detail
Chapter 5	This chapter shows the web portal for proposed framework and the results
Chapter 6	This chapter is the conclusion of the project about giving an idea of overall review, limitations and discussion on future works to be researched

Chapter 2
Background

2.1 Literature Reviews

2.1.1 Process of Life Science Discovery

Reductionist molecular biology is a hypothesis-based approach used by scientists in the second half of the 20th century to determine and characterize molecules, cells and major structures of living systems. Biologists identified that, as a single community, they are required to continue using reductionist strategies to further their cause in elucidating the whole structure of components and every single one of their functions. They are required to use the system-level type of approach to comprehend molecules and cells, the functions of organs, tissues and populations as well [14]. Other than using information on parts of proteins, genes, and the various other macromolecular entities, systems analysis demands the information on the relationships between molecular parts and how these parts function together [14]. This approach is causing scientists to gradually abandon reductionist approaches while adapting synthetic approaches to identify characteristics and integrate biological data that can be used for quantitative and detailed qualitative predictions in biology systems. Information integration from data sources are heavily depending on a synthetic or integrated view of biology [14].

A hefty amount of research has been done in evolutionary biology since two decades ago. It has highly depended on sequence evaluations at protein levels and of genes. In future work, the approach will grow to be more dependent on tracking DNA sequences and evolution of genomes [14].

Essentially, research discovery enables researchers to obtain complex information from biology and experimental observations of diversity and heterogeneity [14]. Implementation of solid information infrastructures are essential to biology and required in computing activities and databases. An example of how biological research has gradually grown more dependent on integration of computational activities and experimental procedures are shown in Fig. 2.1.

C. Y. Cho et al., *Large Scale Data Analytics*,
Studies in Computational Intelligence 806,
https://doi.org/10.1007/978-3-030-03892-2_2

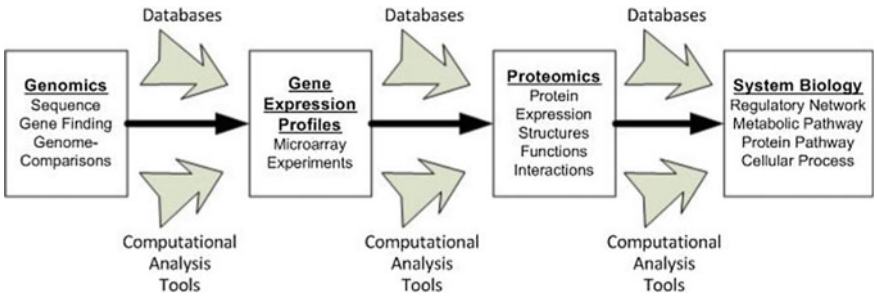


Fig. 2.1 Process of life science discovery

Relations between the area of gene expression profiles, systems biology, proteomics, and genomics are highly dependent on the integration of experimental procedures along with a searchable database, computational algorithm applications and analysis tools [14]. Data from computational analysis and database searches are essential to the whole discovery procedure. Since the selected systems are complex to study, the derived data from simulations and derived computational models obtained from databases are combined to generate experimental data for better interpretations. Studies on protein pathways, cellular and biochemical processes, simulation and modelling of protein-protein interactions, genetic regulatory networks, normal and diseased physiologies are currently in their infancy state, hence, some changes are needed [14]. Quantitative details are missing in the process and experimental observations are needed to fill in the missing pieces. The boundaries between these experimental datasets and computationally generated data are not defined due to close interaction, therefore, multidisciplinary groups are required to integrate these approaches in accelerating progress. With the continuing advances made using experimental methods, information infrastructure can compute the understanding of biology with ease [14].

2.1.2 The Biological Data Nature

As high-throughput technologies are introduced to the biological research field and advanced genome projects, the amount of obtainable data is highly increased and contributed to the large data volume growth as stated by Sidhu et al. [15]. However, data volume is not the focus point in life science. Diversity and variability of data are much more important compared to data volume.

According to Sidhu et al. [15], the structure of a biological dataset is highly complex, and it is organized in a free and flexible hierarchy that reflects the understanding of the complicated living systems involved. These living systems contain information on genes and proteins, regulatory network and biochemical pathways, protein-protein interaction, cells and tissues, ecosystems on earth and

organisms and populations. This raises a series of challenges in modelling, informatics and simulations. There are varieties of biological data due to the complex biological systems ranging from protein and nucleic acid sequences, different levels of biological images resolutions, literature publications and laboratory records, to dozens of technological experimental outputs such as light and electronic microscopy, microarray chips, mass spectrometry as well as results from Nuclear Magnetic Resonance (NMR) [15].

The differences of different types of individual and species varies immensely, as well as the nature of biological data. For instance, the function and structure of organs are different depending on the age and gender, normal or unhealthy state, and the type of species [15]. Biological research is still undergoing an expansion where different fields in biology are still in their growing stages. Data contributed by these systems are still incomplete and inconsistent. This is a challenging issue in the process of modelling biological objects.

2.1.3 Constant Evolution of a Domain

Lacroix [16] mentioned as domains are constantly changing, the Biological Information Systems must be constructed in a way where handling data is possible while managing the technology and legacy data. Existing data management methodologies are unable to address the constant changes in these domains. There are two major problems that need to be addressed in scientific data management which are changes in data identification and data representation [16].

2.1.3.1 Traditional Database Management

There are three varieties widely used in traditional data management systems. These varieties are relational, object-relational, and object-oriented. According to Lacroix [16], data in relational database systems are represented in a form of relations table with data representation through classes relying on a basic relational representation provided by object-relation systems. The data representation is user-friendly as data are organized through classes as well for object-oriented databases. Traditional database systems are made to support their own data transactions, however, there is a limitation in data changes that can be supported by the data organization of the database. For example, the changes are limited to renaming, adding or removing attributes and relations, and other particulars. Complex schema transactions are not supported by traditional database systems as the initial designs did not take them into account. To define a new schema, a new database will need to be constructed. This will bring changes in the data organization of the database. From a biological data source standpoint, the said process is too troublesome and unacceptable when changes have to be frequently made [16].

From another aspect, traditional database systems depend heavily on pre-defined identities. The set of attributes are primary keys that identifies objects and places them in a relational database. As biological data source attributes are ever changing, the existing concept is not efficient due to the fact that the primary keys do not change over time [16]. There is no biological data management system designed to keep up with the frequent changes in identification, such as tracking the frequent changes of identity in objects.

2.1.3.2 The Fusion of Scientific Data

Data fusion defines an implementation of data that are obtained from different types of sources. Scientific data are obtained from different instruments performing mass spectrometry, microarrays and other specific procedures [16]. These instruments rely on proper calibration parameters setup for standardized data collection. Data collected from similar tasks performed on these instruments can be implemented into the same dataset for analysis.

Using a traditional database approach, complete dataset measurements and parameters are required for complex queries for the data analysis process [16]. If any information is missing or incomplete, the data will be ignored and left unprocessed, which is unacceptable to life data scientists.

2.1.3.3 Differences of Structured and Semi-structured Data

The integration of datasets that are alike but disparate in the biological domain is not supported by existing traditional database methodologies. The solution for this problem is to adhere to the structure offered by semi-structured methods [16]. A feature where data organization enables the changes of new attributes and missing attributes are introduced in this semi-structured method. Semi-structured data is usually shown as either rooted, edge-labelled or directed graph. XML is one of the examples of semi-structured data. XML has become the standard for storing, describing and interchanging data between many heterogeneous biological databases [16]. The facilities for XML content definition are provided by the combination of multiple XML schemas [16]. Flexibility and platform support that are ideal for capturing and representing the complicated data types of biological data can be provided by XML.

2.1.4 Data Integration Challenges

Data Integration was never easy to begin with. Researchers are struggling to improve data integration processes to ensure that data translation can be done in a fast and efficient manner. Kadadi et al. [17] had conducted a survey on the

challenges of data integration and interoperability in large scale data and summarized these challenges into 7 parts: accommodation for scope of data, data inconsistency, query optimization, inadequate resources, scalability, implementing support system and Extract Load Transform (ETL) processes in big data. The challenge to accommodate the scope of large datasets and the addition of new domains in any organization can be overcome by integrating high performance computing (HPC) environments and high-performance data storage, for example, hybrid storage devices with the combined functionality of a standard hard disk drive (HDD) and solid state drive (SSD) to reduce data latency and to provide fast data access. However, this method leads to the need to upgrade or purchase new equipment.

In a survey conducted by Kadadi et al. [17], they clarified that data from different sources leads to data inconsistency, thus high computing resources are needed to process unstructured data from large data sources. Therefore, query operations are easier to perform on structured data to analyse and obtain data for various uses, such as business decisions. However, in large datasets, there is normally a high volume of unstructured data. By referring to the survey conducted, query optimization may affect the attributes when data integration takes place at any level or during data mapping to existing or new schema [17].

Furthermore, Kadadi et al. [17] surveyed where problems arise with inadequate resources in data integration implementation; these problems include insufficient financial resources and insufficient skilled personnel in data integration. They also mentioned high level skilled personnel in big data are hard to find and these skilled personnel requires a high level of experience at dealing with data integration modules. Furthermore, the process of obtaining new licenses for tools and technologies from vendors required for data integration implementation is tedious.

Kadadi et al. [17] identified that scalability issues occurred in scenarios where new data are extracted and integrated from different sources along with legacy systems data. Attempting this heterogeneous integration may affect the performance of the system due to the need to undergo updates and modifications for the system to adapt to newer technologies. However, if legacy systems meet the requirements and are compatible with newer technologies, the process is easier as less updates and modifications are necessary in the ensuing integration process.

Support systems need to be implemented by organizations to handle updates and report errors in every step of the data integration process. In the survey conducted by Kadadi et al. [17], they discovered that implementing support systems will require a training module to train professionals on error report handling, and this will require a huge sum of investment for organizations. However, through the implementation of support systems, organizations can determine the weaknesses existing in their system architecture.

Extract Load Transform (ELT) is an example of data integration. ELT processes every piece of data that goes through it and outputs these data as a huge dataset entity after the integration process. The identification of the ELT processes takes place after the data integration process to determine whether it would affect functionality of database storage due to storing huge data chunks [17]. To improve load

processes, key constraints are disabled during the load processing part and re-enabled after the process is done, a step required to be done manually as suggested by researchers [17].

2.1.5 Semantic Integration Challenges

In semantic integration, concepts of interest are defined as a common meta-model, and the properties of data sources are portrayed as common concepts [18]. The system manages data sources while users interact with data mapping. Despite the significance and usefulness of semantic integration, it still has flaws that are difficult to solve. Doan and Halevy [19] had conducted a survey on challenges of semantic integration and these challenges are hard to address due to several fundamental reasons: Involved elements of semantics can only relate to few information sources, the data creators, related schema, documentation and the data itself. Semantic information is difficult to extract, especially from the data creators and documentation. Doan and Halevy [19] stated in the survey that data creators of older databases are likely retired, have moved or have forgotten about their created data. Moreover, any documentation is likely to be untidy, incorrect or outdated. This is a huge problem as the process of matching schema elements is normally done based on the clues between schema and data, for example, the name of the elements, structures, values, types and integrity constraints. Doan and Halevy [19] clarified that these clues are not always reliable as elements might have the same name but can be two different entities, and they are often incomplete. For example, an element with the name contact-agent implies that it is related to the agent but does not provide any substantial information to justify the meaning of the relationship; it could be the agent name or phone number. In the scenario brought up in the survey conducted by Doan and Halevy [19], to match an element s from schema S with element t from schema T, all the other elements in schema T needs to be examined to ensure element t can be represented with s. To further complicate matters, the overall matching process is dependent on the application used. Doan and Halevy [19] suggested users oversee the matching process to avoid any mismatches but user opinion is subjective and this leads to the need of assembling a committee to determine whether the mapping process is correct.

Due to these challenges, semantic matching needed to be done manually and has been long known to be error-prone. For example, 0069n a case where GTE telecommunications attempted to integrate 40 databases with 27,000 elements, the planners for this project estimated that it will take 12 person-years to find documentation and element matches without the original developers of the databases [19].

2.1.6 Biomedical Ontologies

The existing methodologies do not discuss the complex issues of biological data. Recent efforts made on ontologies intended to provide a way to solve these complex problems. According to Gruber [20], the term ontology originates from a philosophical term referring to 'the object of existence' and from the computer science community's perspective, it is known as 'specification of conceptualization' for sharing information in artificial intelligence. A conceptual framework is delivered by ontologies for a significant structured image through common vocabulary provided by biological or medical domains [21]. These can be used by either automated software agents or humans in the domain. The concepts, relationships, definition of relationships and the prospect of ontology rules and axiom definitions are included by shared vocabulary to define the mechanism used to control the substances which are introduced into the ontology and applicable on logical inference [21]. Ontologies are slowly emerging as a common language in biomedicine for higher effective communication needed across multiple sources involving information and biological data.

2.1.6.1 Biomedical Ontologies Open Issues

Researchers tends to select different types of organisms depending on their research work in different fields of biological systems as they progress on their research. For instance, to study human heart disease, the rat is chosen as it is a good model to study. Meanwhile, to study cellular differentiation, the fly is chosen for the task. Each of the model systems consists of paid database overseers collecting and storing biological data for the specific organism [21]. A list of keywords is generated by scientific text mining that are used as the terms for gene ontology. Different terms might be used by different database projects referring to the same theory or concept and sometimes the same term might be referring to a completely different concept. However, these terms might not relate to each other formally in any possible way [21]. To tackle this problem, organized and precise vocabularies are provided by Gene Ontology (GO) and can be shared between biological database to define the gene products. Whether it is from a different or the same database, this enables the querying process of gene products to be performed more easily through information sharing of biological characteristics.

The application of GO links up ontology nodes and proteins, especially for protein annotation over gene ontology. The GO Consortium developed a software platform named GO Engine through the combination of harsh sequences of homology comparisons with the analysis of text information to annotate proteins efficiently [21]. There are new genes forming during evolution created through mutation, recombination with ancestral genes and duplication. Whenever one of the species evolves, high levels of homology will be retained in most of the orthologs.

In biomedical literature, individual gene and protein associated text information is buried deeply among the other biomedical literature. There are few papers published recently describing the growth of numerous methods to extract text information automatically. However, direct implementation of these methods in GO annotation are insignificant [21] but with GO Engine, it can gather homology information, analyse text information and unique procedures of protein-clustering to construct the finest annotations possible.

In recent events, Protein Data Bank (PDB) has also released a few versions of PDB Exchange Dictionary and its archival files in the format of XML, namely PDBML. Both XML Representations and PDB Exchange uses similar logical data organization but disadvantages of being able to maintain a rational communication with PDB Exchange representation is PDBML lacking categorized structure properties in XML data. Ontology induction tool, a directed acyclic graph (DAG) was introduced to build protein ontology including MEDLINE abstracts and UNIPROT protein names. It represents the relationship between protein literatures and knowledge on protein synthesis process. Nevertheless, the process is not formalized, thus, it can't be recognized as a protein ontology [21].

At the completion of the Human Genome Project (HGP) in April 2003, Genomes to Life Initiative (GTL) was announced [21]. Ongoing management and the coordination of GTL are guided from the experience from HGP states the objective, "To correlate information about multiprotein machines with data in major protein databases to better understand sequence, structure and function of protein machines". The objective can be achieved up to certain extent by constructing the Generic Protein Ontology based on the Generic Protein Ontology vocabulary for proteomics and Specialized Domain Ontologies for every main protein family [21].

2.1.7 Creation of Ontology Methodologies

A new 'skeletal model' was presented by Uschold and King [22] as a design and evaluation for ontologies. There are several stages in the skeletal model that are essential for any ontology engineering related methodology. There are several specific principles designed by Uschold and Gruninger [23] to be uphold in each phase, which are: coherence (consistency), extensibility, clarity, minimal ontological commitment, and minimal encoding bias [20]. A semi-informal ontology named The Enterprise Ontology has been created by Uschold et al. [24] by following the design principles mentioned above for ontology capture phase.

Based off the experiences of creating the TOVE (TOronto Virtual Enterprise) ontology, Gruninger and Fox [25] developed a new methodology for both design and evaluation for ontologies. However, this methodology was designed base on a very rigid method, hence, this methodology is not suitable for any less formal ontologies. Furthermore, this methodology is not sufficient for a first-order logic based ontology language as a first-order logic language is used for this methodology for the formulation of axioms, definitions and its justification.

A methodology presented by Staab et al. [26] was created based on the On-To-Knowledge (OTK), which is a primary key point in constructing large Knowledge Management systems. In the methodology presented, the differences of both knowledge process and knowledge meta-process is made clearly. The knowledge process is responsible for Knowledge Management system which deals with knowledge acquisition and retrieval while knowledge meta-process deals with managing the knowledges in the system. In ontology terms, the first part of the methodology deals with the usage of ontology while the latter deals with the initial set up, construction and maintenance of the ontologies.

The skeletal model by Uschold and King is not a methodology, but rather a standard to be followed by ontology engineering related methodologies. Meanwhile, the methodology presented by Gruninger and Fox is only suitable for formal logic languages and it was specifically created using KIF language [27, 28]. This methodology was tailored for formal authentication through the usage of formal questions. However, the approach of METHONTOLOGY is much more generic and a comprehensive methodology compared to the others. METHONTOLOGY offers a generic methodology for all types of ontology while obeying the standard of IEEE software development process. The On-To-Knowledge methodology can give a better support for ontology developer as it is built specifically for the development of both domain and application related ontologies, which is the Knowledge Management applications for ontologies.

2.1.7.1 The Creation of Protein Ontology with On-To-Knowledge Methodology

The On-To-Knowledge methodology has two main approaches for Knowledge Management during the creation of Protein Ontology:

1. Data Focus: Mainly pragmatic, the data focus approach has been chosen by organizations that maintain protein data in Knowledge Management; to review the current protein databases and identifies the knowledge needs. Meta-data is defined as “Data that describes the structure of data” in data focus.
2. Knowledge Item Focus: For the knowledge item focus, the established knowledge of Protein Ontology classifies knowledge needs through the examination of knowledge items. The meta-data for knowledge item focus is defined as “Data describing issues related to the content of data”.

Once the implementation of knowledge management system for Protein Ontology has been done, the knowledge processes cycle through these few steps, which are also illustrated in Fig. 2.2 [26]:

1. The process of creating and importing Protein Data from different data sources.
2. Gathering knowledge related to the concepts of protein ontology, including protein data annotation and the references of protein ontology concepts.

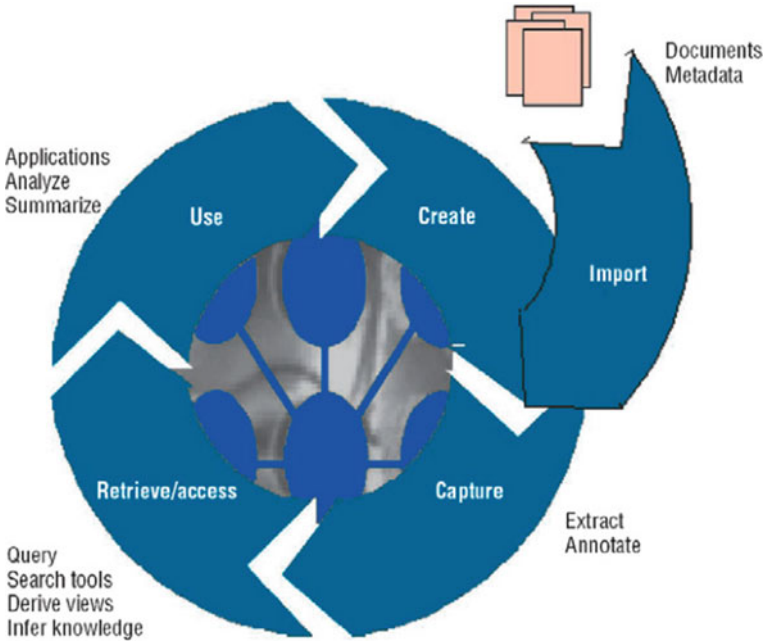


Fig. 2.2 Process of On-To-Knowledge

3. The process of retrieving and accessing knowledge from the concepts of protein ontology using a query.
4. User goals are achieved through the usage of extracted knowledge.

There are several phases in the process of protein ontology development using On-To-Knowledge methodology as shown in the Fig. 2.3 [29]:

1. Phase one of the process is the feasibility study. This phase is implemented from the CommonKADS methodology [30]. CommonKADS is a framework used to develop a knowledge-based system (KBS) and it supports the features of KBS development project, for example: acquisition of knowledge, problem identification, project management, knowledge modelling and analysis, system integration issues analysis, capturing user requirements, and knowledge system design. The outcome, that has been determined after conducting the feasibility study, was that On-To-Knowledge should be used to construct Protein Ontology for maximum support on its development, maintenance and evaluation.
2. Phase two, which is the actual first phase in development, outputs the ontology requirement specification. The possibilities of having existing protein data sources integrated into the ontology are analyzed in this stage. In addition, there are a number of queries generated to capture the protein ontology requirements for existing protein data and knowledge frameworks.

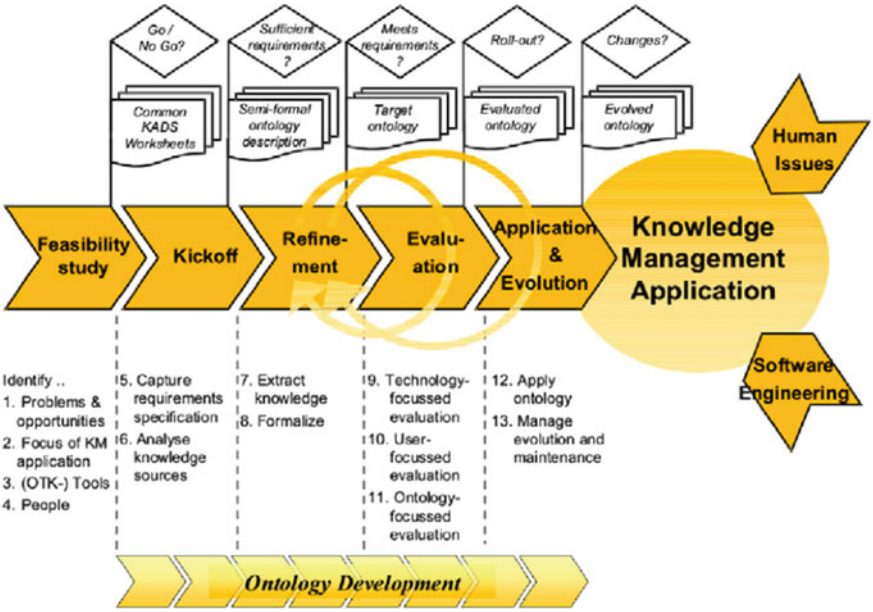


Fig. 2.3 Ontology development with On-To-Knowledge

3. In phase three, which is the refinement phase, proteomics domain-oriented protein ontology is developed based on the specification received from phase two. There are several sub phases for this phase where:
   a. Baseline taxonomy was gathered for Protein Ontology.
   b. Seed ontology was created according to the baseline taxonomy which has the related protein data concepts and descriptions for the protein data relationships.
   c. Target protein ontology was then generated through the usage of seed ontology and expressed in the form of a formal language, Web Ontology Language [31].
4. Phase four, the evaluation phase is the final phase of the ontology development stage. During this phase, the specification document and queries are used to verify the protein ontology. The usage of protein ontology in proteomics domain is evaluated in this phase as well. Feedback gathered from different research teams that are using the protein ontology during the evaluation phase is processed in the refinement phase. Through this method, the process will go through several cycles until the protein ontology has been verified to be acceptable for usage.

5. Phase five, the maintenance phase is engaged after the protein ontology has been deployed. In this phase, all the changes that occur in the world will reflect onto the protein ontology.

2.1.8 Ontology-Based Approach for Semantic Integration

An approach of semantic information integration done by Ngamnij and Somjit [32] for electronic patient records (EPR) using an ontology and web service model by using ontology for mapping purposes, data extraction, data translation and data integration [32]. The concept of their system is to integrate data from various healthcare institutes into a single database to ease the data retrieval process. In their framework [32], Semantic Bridge Ontology Mapping was used to map web services descriptions and databases of healthcare institutes in WSDL format. The data was then used to construct Ontology-based Patient Record metadata (OPRM). OPRM data needs to be translated and stored via a Domain Ontology Extraction and Semantic Patient Record Integration process [32]. Domain Ontology Extraction converts information of patient from each healthcare system and convert these records to OWL (Web Ontology Language) format using Jena API [33], a Java open source Semantic Web framework. Semantic Patient Record Integration then stores the data into a single database [32] that maps the description of multiple EPR. An EPR Semantic Search allows users to retrieve information from the stored OPRM, for example, the type of medical treatment a patient is receiving. This is the approach made by Ngamnij and Somjit [32] as an alternative way to tackle the multiple data sources querying issue.

Liu et al. [34] introduced a semantic data integration approach with domain ontology, OPSDS. OPSDS is used widely in multiple platforms of China Petroleum Corporation (CNPC) and it is introduced to integrate oil production engineering data. This methodology introduced by Liu et al. [34] focuses on data integration using domain-oriented method, enabling users to access data and shared service through the usage of transforming query, mapping of ontology and data cleaning. The approach of Liu et al. [34] methodology is to build a system where users and applications can access data at ease with the assists of a well-equipped semantic view for underlying data. Figure 2.4 shows the OPSDS architecture.

The bottom layer of the architecture as shown in Fig. 2.4 are different databases containing different data sources, for example, SQL Server, Oracle, and other databases. The middle layer of the architecture consists of local ontologies mined from the various data sources from the bottom layer. Therefore, the group of local ontologies combined and formed a unified global ontology. With this architecture, Liu et al. [34] mentions where users and applications can retrieve data easily by querying the global ontology.

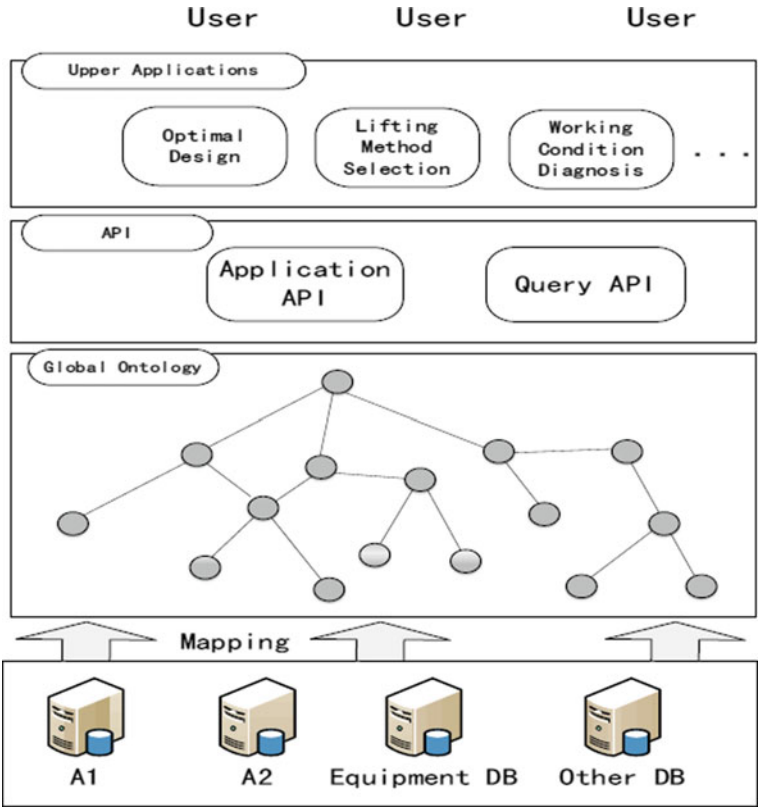


Fig. 2.4 OPSDS architecture

The focus of the architecture is the global ontology. Liu et al. [34] way of constructing a global ontology is through adapting a hybrid strategy. Figure 2.5 shows the process of global ontology.

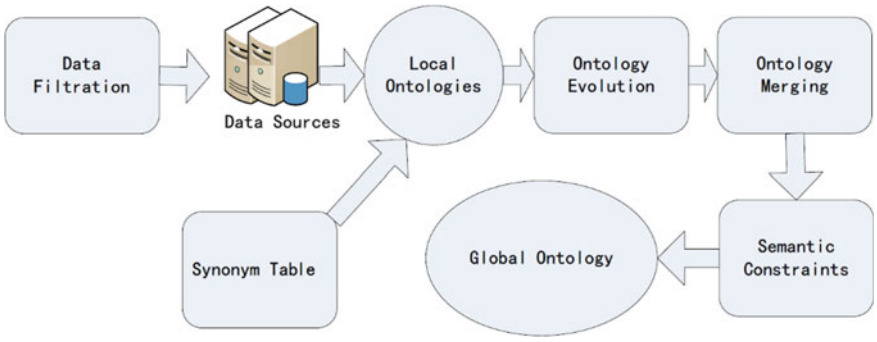


Fig. 2.5 Process of global ontology

The first phase of the global ontology process is filtering data from different data sources, such as entities of the data, relationships and attributes. The second phase of the process is to generate local ontologies through retrieving schemas from databases and items from the synonym table. The global ontology process is completed through ontology evolution, mapping and applying semantic constraints [34].

In summary, the existing methodologies are focusing on integrating multiple data sources into a single data source and applying ontology-based semantic integration as a solution to the problem of data query for multiple data sources. Existing methodologies can be used for integrating small amount of data, however, not for petabytes of data. Taking RCSB PDB as an example, RCSB RPD databases are updated from time to time and it is hard and expensive for these methodologies to live update their database while mapping data at the same time. Multiple data integration challenges are not properly addressed even with semantic integration and ontology-based semantic integration approaches.

In this research, the focus is on querying data sources with different data structures without the need of data integration and data translation. Therefore, the implementation of a smart query system using Language Integrated Query is required to reach the research goal.

Chapter 3
Large Scale Data Analytics

The nature of protein data is complicated and constantly updated by researchers around the globe. To query from multiple data sources, a query framework written and built using Python with the concept of Language Integrated Query is proposed as the solution to overcome the limitations discussed in previous chapters. A cloud computing platform is used for this research to host the query framework to enable the framework to use the vast resources available to perform a query with minimal latency while avoiding computing resource deficiency. In this chapter, Language Integrated Query, cloud computing and algebraic operators are explained in detail.

3.1 Language Integrated Query

Traditional type of queries is expressed in simple string instead of having type checking during compilation or IntelliSense support. To query databases, different query languages need to be studied and understood to use each data source with differing data structures, such as SQL databases, variable Web services, XML documents and others [35].

Language integrated query bridges both worlds of data and object. It was first introduced in Visual Studio 2008 and .NET Framework version 3.5 [35]. Language integrated query can be written in Python, C# or Visual Basic in Visual Studio and it is compatible with SQL Server databases, ADO.NET datasets and XML documents [35]. This method can be applied in new projects and existing projects. Query writing is easier and better through the usage of keywords of the language and by using familiar operations with typed collections of objects.

Parallel Language Integrated Query is an engine included in .NET framework version 4 and it is used to execute queries in a parallel manner. This execution of queries can be sped up efficiently through the usage of computing resources provided by the host computer and this feature relies heavily on the host computer itself, in this case, the cloud computing platform. Another major component for

C. Y. Cho et al., *Large Scale Data Analytics*,
Studies in Computational Intelligence 806,
https://doi.org/10.1007/978-3-030-03892-2_3

Language Integrated Query is the ability to query across relationships. This approach enables users to query through accessing properties of a relationship and to navigate from one object to another [36]. The access operations are transformed into a complex join or corresponding sub queries in an alternative SQL [36].

For this research, Language Integrated Query written using Python is more convenient. Python is a popular high-level programming language widely used for Rapid Application Development which has the functions of object-oriented and dynamic semantics [37]. A vast standard library and interpreter of Python are obtainable in binary or source code and available to all platforms for free. It is widely used for Rapid Application Development, scripting or connecting existing modules together due to python's effectiveness in data structures, dynamic binding and typing. Through the usage of python, program maintenance cost is lower as the language itself is simpler and easier to learn compared to other languages. Its modules and package capabilities widely support the idea of modular programming and reusing of code [37]. Programmers using Python are not required to compile their programs, an essential process in all other major programming language currently available, making the process of editing to debugging cycle more efficient [37].

3.2 Cloud Computing as a Platform

The framework is deployed on a cloud computing platform, Microsoft Azure, to allow the framework to operate smoothly using its vast computing resources and to benefit from the much lower operating cost compared to having on-site hardware.

Cloud computing can be defined as the use of hosted services through the internet. The 'Cloud' moniker came from the flowchart or cloud-shaped diagram by which the internet was generally represented [38]. Cloud computing has been utilized by users over the world to gain advantages over current technologies. The operational model changes the initial impression of needing to store applications in physical hardware to the impression that it is unnecessary to store these applications in physical hardware. Due to its flexibility, the computational resources can be changed easily depending on the demand of users [38]. The available cloud services are ready to be used without the need of great knowledge or skills to deploy these services [39]. Services are ready to be deployed and can be done over the internet helping to cut the cost required to hire professional personnel for the task and this helps with the financial situation of any company.

3.3 Algebraic Operators for Biomedical Ontologies

3.3.1 Select Operator

The projection over sequence is performed by the Select Operator. The allocation and return of the enumerable object done by the Select Operator captures the arguments passed to the operator. An argument null exception is returned if any argument is null [40].

The Select operator allows the user to highlight and select the portions of an ontology related to the user's query. The Select Operator selects the instances meeting the condition given through the ontology structure and the selected concept given. These instances, which met the given condition, would belong to a specific sub tree or are the subset of the instances that belong to one or more sub trees. The Select Operator selects only those edges in the ontology that connect nodes in each set. The Select Operator, OS is defined as:

Definition 1

$$\mathrm{OS} = \sigma(\mathrm{NS}, \mathrm{ES}, \mathrm{RS}) \text{ where}$$
$$\mathrm{NS} = \mathrm{Nodes}(\mathrm{condition} = \mathrm{true})$$
$$\mathrm{ES} = \mathrm{Edges}(\forall \mathrm{N} \in \mathrm{NS})$$

N, E, R here are represented as set of nodes, edges and the relationships of the ontology graph while NS, ES, RS are presenting the nodes, edges and relationships of the set selection. The join condition operator won't be discussed here as the Select Operator can be used in the following forms:

- Simple-Condition: Where the select condition is specified using the simple content types, like Generic Concepts, in the ontology and the select operator is value-based;
- Complex-Condition; Where the select condition is specified using complex content types, like Derived Concepts, in the ontology and the select operator is structure-based; and,
- Pattern-Condition: Where the select condition is specified using a mix of simple and/or complex content types in the hierarchy with additional constraints such as ordering defined by using of Sequence Relationships in the ontology and others, where the select operator is pattern-based.

Example 1
When the user requires every available information in Protein Ontology in respect to the Protein Families, the details of every single example of the Family Concept is displayed by using the Select Operator which is shown in Fig. 3.1.

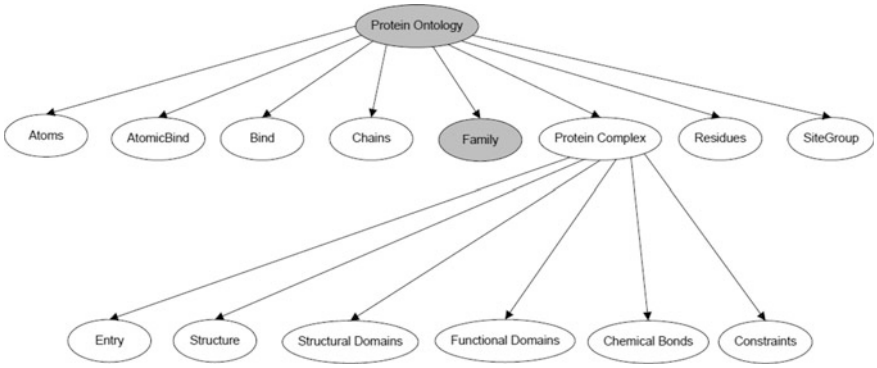


Fig. 3.1 Usage of select operator in instances of family concept

3.3.2 Union Operator

The union set between two sequences is produced by the Union Operator. The allocation and returns of the enumerable object which is done by the Union Operator captures the arguments which are passed on to the operator. An argument null exception is returned if any argument is null [40].

When Union returns the enumerated object, first and second sequences are enumerated, in that particular order, and will yield onto each element which was not previously yielded. Elements are compared by using the non-null comparer argument if possible. Otherwise, the equality comparer is utilized.

The union of two parts of the ontology, O1 = (N1, E1, R1), and O2 = (N2, E2, R2) with respect to the semantic relationships (SR) of the ontology is expressed as:

Definition 2

$\mathrm{OI}(1,2) = \mathrm{O1} \cup_{\mathrm{SR}} \mathrm{O2} = (\mathrm{NU}, \mathrm{EU}, \mathrm{RU})$, where,
$\mathrm{NU} = \mathrm{N1} \cup \mathrm{N2} \cup \mathrm{NI}(1,2)$
$\mathrm{EU} = \mathrm{E1} \cup \mathrm{E2} \cup \mathrm{EI}(1,2)$, and
$\mathrm{RU} = \mathrm{R1} \cup \mathrm{R2} \cup \mathrm{RI}(1,2)$, where,
$\mathrm{OI}(1,2) = \mathrm{O1} \cap_{\mathrm{SR}} \mathrm{O2} = (\mathrm{NI}(1,2), \mathrm{EI}(1,2), \mathrm{RI}(1,2))$ is the intersection of two ontologies.

Two parts of the ontology are combined by the union operation and only one copy of the intersection concepts is retained. N, E, R here are represented as set of nodes, edges and the relationships of the ontology graph while NU, EU, RU are representing the nodes, edges and relationships of the set selection.

Example 2
When a person requires all the available information in Protein Ontology in respect to the protein Structure and Protein Families, every single information which are highlighted in Fig. 3.2 is then output. That is how the Union Operator is used (Family $\cup$ Structure).

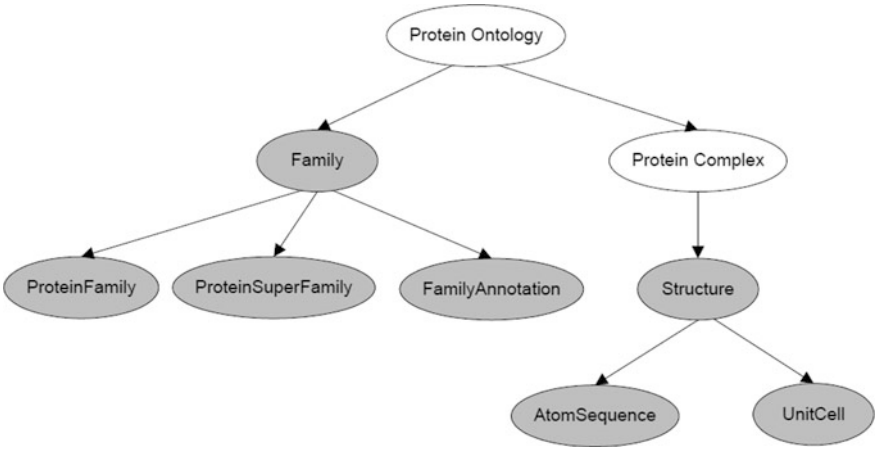


Fig. 3.2 Usage of union operator

3.3.3 Intersection Operator

The intersection set between two sequences is produced by the Intersect Operator. The allocation and returns of the enumerable object which is done by the Intersect Operator captures the arguments which are passed on to the operator. An argument null exception is returned if any argument is null [40].

When Intersect returns the enumerated object, the first sequence is enumerated, all the distinct elements of the sequence are collected. The second sequence is enumerated, marking all elements that occur in both sequences. The marked elements are yielded in the manner of how they were collected. Elements are compared by using the non-null comparer argument if possible or using the equality comparer.

Intersection is a particularly significant and fascinating binary operation. There are two parts, O1 = (N1, E1, R1), and O2 = (N2, E2, R2) in the ontology whereas an answer to the query submitted is provided by the composition of both ontologies. N, E, R here are represented as set of nodes, edges and the set of Semantic Relationship. The ontology semantic relationships in respect to the intersection of two parts of the intersection operation is:

Definition 3

$$OI(1,2) = O1 \cap_{SR} O2 = (NI, EI, RI), \textit{ where}$$
$$NI = Nodes(SR(O1, O2)),$$
$$EI = Edges(E1, NI \cap N1) + Edges(E2, NI \cap N2) + Edges(SR(O1, O2)), \textit{ and}$$

$$RI = Relationships(O1, NI \cap N1) + Relationships(O2, NI \cap N2) + SR(O1, O2) - Edges(SR(O1, O2)).$$

SR is totally different compared to R since that it does not include sequences in it. The nodes which are in the intersection ontology are the nodes which exists in semantic relationship, which is represented by SR. The intersection ontology edges among the nodes are either already existing in the ontology sources or has been recognized as SR. The connections of the intersection ontology are the ones that have still not been modeled as the edges. The connections which are existing in the ontology sources only use the concepts that are happening in the intersection ontology.

Example 3
When a query needs all the available information which are common between the Protein Structure and the Protein Entry descriptions in Protein Ontology, the only common thing in between both is the **ChainsRef**. As shown in Fig. 3.3 that is how the Intersection Operator is used (Entry $\cap$ Structure).

3.3.4 Except Operator

The differences of both two sequences is produced by the Except Operator. The allocation and return of the enumerable objects which is done by the Except Operator captures the arguments which are passed on to the operator. An argument null exception is returned if any argument is null [40].

When Except returns the enumerated object, the first sequence is enumerated, and all the distinct elements of that sequence are collected. The second sequence is enumerated and the elements which resides in the first sequence is deleted. Then in order, the remaining elements are finally yielded in the way they were collected. Elements are compared by using the non-null comparer argument if it is possible. Otherwise, the equality comparer is utilized.

The differences between O1 and O2, which are the two parts of the ontology are presented as O1 − O2 which includes portions from the first part which are not the common in the second part. The difference can also be represented as $O1 - (O1 \cap_{SR} O2)$. Nodes, edges and relationships that are not present in the intersection, but exists in the first ontology.

Example 4
When a query needed all the available information on Protein Entry without the Protein Structure and Protein Entry descriptions which resides in Protein Ontology, every single information of Protein Entry that is not been highlighted in the previous Fig. 3.3 is displayed. As ChainsRef is the only common in between both Protein Structure and Protein Entry, everything else excluding ChainsRef is output

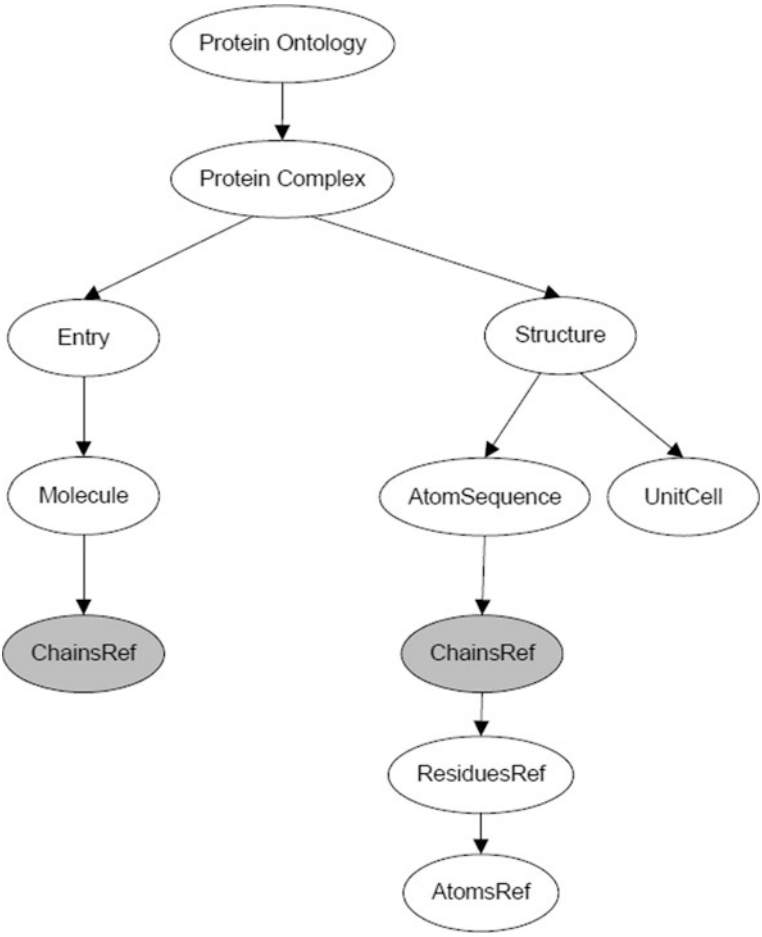


Fig. 3.3 Usage of intersection operator

for the Protein Entry by using the Difference Operator (Entry—(Entry ∩ Structure)). The objective of having to compute the differences is to optimize the Protein Ontology maintenance.

The instance of Protein Ontology storage is huge and there are a lot of user constantly adding instances to it. The differences will expose the instances that have not been keyed in properly or if there are any changes to the data sources which are being integrated by Protein Ontology. The changes which are uncovered by the differences are forwarded to the administrator.

Therefore, the Semantic Relationships do not need to be modified or changed. If changes arise from the changes to the data source which was integrated by Protein Ontology, then the semantic relation and the concepts need to be clarified for any further changes needed to remove the difference.

Chapter 4
Query Framework

4.1 Functions for Querying RCSB Protein Data Bank (PDB)

Protein Data Bank, PDB has a vast amount of resources related to protein 3D models, complex assemblies, and nucleic acids that can be utilized by both students and researchers for learning the characteristics of biomedicine. Therefore, a framework is needed to effectively retrieve information from their database. The functions that are utilized to enable users to query RCSB PDB is explained in this chapter.

4.1.1 Make Query Function

Figure 4.1 shows the structure and python codes constructed using Visual Studio for make query function.

The make_query() function initiates a search based on a list of search terms and requirements and outputs as a compiled dictionary object which users can search later on. There are several query types that can be used for the search, which are as follows:

HoldingsQuery	A normal search of any related PDB IDs metadata.
ExpTypeQuery	A search based on experimental method, for example, 'X-RAY'.
AdvancedKeywordQuery	Any matches that appears in either the title or abstract.
StructureIdQuery	A normal search by provided structure ID.
ModifiedStructuresQuery	Search based on the structures relevancy.
AdvancedAuthorQuery	A search on entries based on the name of author.
MotifQuery	A normal search for motif.
NoLigandQuery	Search every PDB IDs that has no free ligands

C. Y. Cho et al., *Large Scale Data Analytics*,
Studies in Computational Intelligence 806,
https://doi.org/10.1007/978-3-030-03892-2_4

```
def make_query(search_term, querytype='AdvancedKeywordQuery'):

    assert querytype in {'HoldingsQuery', 'ExpTypeQuery',
                         'AdvancedKeywordQuery','StructureIdQuery',
                         'ModifiedStructuresQuery', 'AdvancedAuthorQuery', 'MotifQuery',
                         'NoLigandQuery'
                        }

    query_params = dict()
    query_params['queryType'] = querytype

    if querytype=='AdvancedKeywordQuery':
        query_params['description'] = 'Text Search for: '+ search_term
        query_params['keywords'] = search_term

    elif querytype=='NoLigandQuery':
        query_params['haveLigands'] = 'yes'

    elif querytype=='AdvancedAuthorQuery':
        query_params['description'] = 'Author Name: '+ search_term
        query_params['searchType'] = 'All Authors'
        query_params['audit_author.name'] = search_term
        query_params['exactMatch'] = 'false'

    elif querytype=='MotifQuery':
        query_params['description'] = 'Motif Query For: '+ search_term
        query_params['motif'] = search_term

    # search for a specific structure
    elif querytype in ['StructureIdQuery','ModifiedStructuresQuery']:
        query_params['structureIdList'] = search_term


    elif querytype=='ExpTypeQuery':
        query_params['experimentalMethod'] = search_term
        query_params['description'] = 'Experimental Method Search : Experimental Method='+ search_term
        query_params['mvStructure.expMethod.value']= search_term


    scan_params = dict()
    scan_params['orgPdbQuery'] = query_params


    return scan_params
```

Fig. 4.1 Make query function

As an example, a search based on ‘actin network’ will return a result of ‘1D7M’, ‘3W3D’, ‘4A7H’, ‘4A7L’, ‘4A7N’.

4.1.2 Do Search Function

Figure 4.2 shows the code and structure in python used for do search function.

The function do_search() converts the dictionary, dict() object into XML format which then sends a request to obtain a matching list of IDs according to search results from PDB. In this case, the results obtained from make_query() function are converted to XML format and the XML format will prompt PDB for a list of matching PDB IDs.

```
def do_search(scan_params):

    url = 'http://www.rcsb.org/pdb/rest/search'

    queryText = xmltodict.unparse(scan_params, pretty=False)
    queryText = queryText.encode()

    req = urllib.request.Request(url, data=queryText)
    f = urllib.request.urlopen(req)
    result = f.read()

    if not result:
        warnings.warn('No results were obtained for this search')

    idlist = str(result)
    idlist =idlist.split('\\n')
    idlist[0] = idlist[0][-4:]
    kk = idlist.pop(-1)

    return idlist
```

Fig. 4.2 Do search function

4.1.3 Do Protsym Search Function

Figure 4.3 shows the code and structure of do protsym search function.

The function do_protsym_search() searches identical entries from user-specified symmetry groups in Protein Data Bank, PDB. The total minimum and maximum deviation allowed is measured in Angstroms, are adjusted to determine which results will be categorized as an identical symmetry. For instance, when 'C9' has been used as the point group, the results returned are shown as '1KZU', '1NKZ', '2FKW', '3B8M', '3B8N' respectively.

```
def do_protsym_search(point_group, min_rmsd=0.0, max_rmsd=7.0):

    query_params = dict()
    query_params['queryType'] = 'PointGroupQuery'
    query_params['rMSDComparator'] = 'between'

    query_params['pointGroup'] = point_group
    query_params['rMSDMin'] = min_rmsd
    query_params['rMSDMax'] = max_rmsd

    scan_params = dict()
    scan_params['orgPdbQuery'] = query_params
    idlist =  do_search(scan_params)
    return idlist
```

Fig. 4.3 Do protsym search function

Because of the staggering number of components expected on exascale computing systems, hardware failures are expected to increase. Traditional error correction methods, such as the checkpoint-restart recovery mechanism, become too expensive in terms of time and energy due to bulk synchronization and I/O with the file systems at exascale. An exascale system that employs global recoveries could conceivably take more time to ready itself than the mean time between failures.

4.1.4 Get All Function

Figure 4.4 shows the code used to construct get all function.

The function get_all() lists out all the currently available PDB IDs in the RCSB Protein Data Bank.

4.2 Functions for Looking up Information Given PDB ID

4.2.1 Get Info Function

Figure 4.5 shows the code and structure of get info function.

The function get_info() retrieves all information related to the inserted PDB ID. By combining the specific URL and PDB ID, information regarding specific protein data can be retrieved.

Fig. 4.4 Get all function

```
def get_all():

    url = 'http://www.rcsb.org/pdb/rest/getCurrent'

    req = urllib.request.Request(url)
    f = urllib.request.urlopen(req)
    result = f.read()
    assert result

    kk = str(result)

    p = re.compile('structureId=\"....."')
    matches = p.findall(str(result))
    out = list()
    for item in matches:
        out.append(item[-5:-1])

    return out
```

```
def get_info(pdb_id, url_root='http://www.rcsb.org/pdb/rest/describeMol?structureId='):

    url = url_root + pdb_id
    req = urllib.request.Request(url)
    f = urllib.request.urlopen(req)
    result = f.read()
    assert result

    out = xmltodict.parse(result,process_namespaces=True)

    return out
```

Fig. 4.5 Get info function

4.2.2 Get PDB File Function

Figure 4.6 shows the structure and codes in Visual Studio of get PDB file function.

For this function, get_pdb_file() allow users to retrieve the full PDB file through inputting a desired PDB_ID. There are a few file types can be retrieved from PDB, namely pdb, cif, xml and structfact. The default selection is set to pdb, however, users can change the file type to their desired one. The compressed (gz) file is retrieved from PDB as well in this process.

4.2.3 Get All Info Function

Figure 4.7 shows the python codes and structure of get all info function.

```
def get_pdb_file(pdb_id, filetype='pdb', compression=False):

    fullurl = 'http://www.rcsb.org/pdb/download/downloadFile.do?fileFormat='
    fullurl += filetype

    if compression:
        fullurl += '&compression=YES'
    else:
        fullurl += '&compression=NO'

    fullurl += '&structureId=' + pdb_id
    # url = 'http://www.rcsb.org/pdb/files/'+pdb_id+'.pdb'
    req = urllib.request.Request(fullurl)
    f = urllib.request.urlopen(req)
    result = f.read()
    result = result.decode('unicode_escape')

    return result
```

Fig. 4.6 Get PDB file function

```
def get_all_info(pdb_id):

    out = to_dict( get_info(pdb_id) )['molDescription']['structureId']
    out = remove_at_sign(out)
    return out
```

Fig. 4.7 Get all info function

The get_all_info() function serves as a wrapper function for get_info() to tidy up results that had been retrieved.

4.2.4 Get Raw Blast Function

Figure 4.8 shows get raw blast codes and structure coded in Visual Studio.

The purpose of get_raw_blast() function is to search the full BLAST page for inserted PDB ID. The BLAST page can be shown in either XML, TXT, or HTML format depending on the preference of the user. The default setting is set to HTML.

4.2.5 Parse Blast Function

Figure 4.9 shows the code and structure written in Visual Studio for parse blast function.

The parse_blast() function is used to clean up retrieved HTML BLAST selection. BeautifulSoup and re module are needed for this function to work. The function processes all complicated results from the BLAST search function and compile matches into a list. A raw text file is shown to display alignment of all matches. HTML type of inputs are much more suited for this function compared to the others.

```
def get_raw_blast(pdb_id, output_form='HTML', chain_id='A'):

    url_root = 'http://www.rcsb.org/pdb/rest/getBlastPDB2?structureId='
    url = url_root + pdb_id + '&chainId='+ chain_id +'&outputFormat=' + output_form
    req = urllib.request.Request(url)
    f = urllib.request.urlopen(req)
    result = f.read()
    result = result.decode('unicode_escape')
    assert result

    return result
```

Fig. 4.8 Get raw blast function

```
def parse_blast(blast_string):

    soup = BeautifulSoup(str(blast_string), "html.parser")

    all_blasts = list()
    all_blast_ids = list()

    pattern = '></a>....:'
    prog = re.compile(pattern)

    for item in soup.find_all('pre'):
        if len(item.find_all('a'))==1:
            all_blasts.append(item)
            blast_id = re.findall(pattern, str(item) )[0][-5:-1]
            all_blast_ids.append(blast_id)

    out = (all_blast_ids, all_blasts)
    return out
```

Fig. 4.9 Parse blast function

4.2.6 Get Blast Wrapper Function

Figure 4.10 shows the code for get blast wrapper function.

The function get_blast2() is an alternative way of searching BLAST with the inserted PDB ID. This function serves as a wrapper function for get_raw_blast() and parse_blast().

4.2.7 Describe PDB Function

Figure 4.11 shows the structure and codes of describe PDB function.

Function describe_pdb() retrieves requested description and metadata for the input PDB ID. For example, details that are shown in Fig. 4.12 for a search includes authors, deposition date, experimental method, keywords, nr atoms, release date, resolution and further related details.

```
def get_blast2(pdb_id, chain_id='A', output_form='HTML'):

    raw_results = get_raw_blast(pdb_id, chain_id=chain_id, output_form=output_form)
    out = parse_blast(raw_results)

    return out
```

Fig. 4.10 Get blast wrapper function

```
def describe_pdb(pdb_id):

    out = get_info(pdb_id, url_root = 'http://www.rcsb.org/pdb/rest/describePDB?structureId=')
    out = to_dict(out)
    out = remove_at_sign(out['PDBdescription']['PDB'])
    return out
```

Fig. 4.11 Describe PDB function

```
citation_authors :          Malashkevich, V.N., Bhosle, R., Toro, R., Hillerich, B., Gizzi, A.,
Garforth, S., Kar, A., Chan, M.K., Lafluer, J., Patel, H., Matikainen, B., Chamala, S., Lim, S., Celikgil, A.,
Villegas, G., Evans, B., Love, J., Fiser, A., Khafizov, K., Seidel, R., Bonanno, J.B., Almo, S.C.
deposition_date :           2013-07-31
expMethod :                 X-RAY DIFFRACTION
keywords :                  TRANSFERASE
last_modification_date :    2013-08-14
nr_atoms :                  0
nr_entities :               1
nr_residues :               390
release_date :              2013-08-14
resolution :                1.84
status :                    CURRENT
structureId :               4LZA
structure_authors :         Malashkevich, V.N., Bhosle, R., Toro, R., Hillerich, B., Gizzi, A.,
Garforth, S., Kar, A., Chan, M.K., Lafluer, J., Patel, H., Matikainen, B., Chamala, S., Lim, S., Celikgil, A.,
Villegas, G., Evans, B., Love, J., Fiser, A., Khafizov, K., Seidel, R., Bonanno, J.B., Almo, S.C.
title :                     Crystal structure of adenine phosphoribosyltransferase from
Thermoanaerobacter pseudethanolicus ATCC 33223, NYSGRC Target 029700.
```

Fig. 4.12 Sample output for describe PDB function

4.2.8 Get Entity Info Function

Figure 4.13 shows constructed codes for get entity info function.

The function get_entity_info() returns all information related to the PDB ID. Information returned to user are entity, type, chain, method, biological assemblies, release date, resolution and the structure ID as shown in Fig. 4.14.

```
def get_entity_info(pdb_id):

    out = get_info(pdb_id, url_root = 'http://www.rcsb.org/pdb/rest/getEntityInfo?structureId=')
    out = to_dict(out)
    return remove_at_sign( out['entityInfo']['PDB'] )
```

Fig. 4.13 Get entity info function

Fig. 4.14 Sample output for get entity info function

```
Entity :
id :            1
type :          protein
Chain :         id :    A
                id :    B
Method :
name :          xray
bioAssemblies : 1
release_date :  Wed Aug 14 00:00:00 PDT 2013
resolution :    1.84
structureId :   4lza
```

4.2.9 Describe Chemical Function

Figure 4.15 shows the code for describe chemical function.

Function describe_chemical() retrieves chemical description of a requested chemical ID. Once the chemical ID, for example, 'NAG' has been selected to retrieve its chemical description, the results returned are shown in Fig. 4.16.

4.2.10 Get Ligands Function

Figure 4.17 shows structure and code constructed in Visual Studio for get ligands function.

```
def describe_chemical(chem_id):

    out = get_info(chem_id, url_root = 'http://www.rcsb.org/pdb/rest/describeHet?chemicalID=')
    out = to_dict(out)
    return out
```

Fig. 4.15 Describe chemical function

```
describeHet :
ligandInfo :
ligand :
molecularWeight :   221.208
InChIKey :          OVRNDRQMDRJTHS-FMDGEEDCSA-N
type :              D-saccharide
chemicalName :      N-ACETYL-D-GLUCOSAMINE
chemicalID :        NAG
smiles :            CC(=O)N[C@@H]1[C@H]([C@@H]([C@H](O[C@H]10)CO)O)O
InChI :             InChI=1S/C8H15NO6/c1-3(11)9-5-7(13)6(12)4(2-10)15-8(5)14/h4-8,10,12-14H,
                    2H2,1H3,(H,9,11)/t4-,5-,6-,7-,8-/m1/s1
formula :           C8 H15 N 06
```

Fig. 4.16 Sample output for chemical function

```
def get_ligands(pdb_id):
    """Return ligands of given PDB ID...
    out = get_info(pdb_id, url_root = 'http://www.rcsb.org/pdb/rest/ligandInfo?structureId=')
    out = to_dict(out)
    return remove_at_sign(out['structureId'])
```

Fig. 4.17 Get ligands function

```
id :                100D
ligandInfo :
ligand:
chemicalID :        SPM
molecularWeight :   202.34
structureId :       100D
type :              non-polymer
InChI :             InChI=1S/C10H26N4/c11-5-3-9-13-7-1-2-8-14-10-4-6-12/h13-14H,1-12H2
InChIKey :          SPERMINE
formula :           C10 H26 N4
smiles :            C(CCNCCCN)CNCCCN
```

Fig. 4.18 Sample output for get ligands function

Function get_ligands() retrieves ligand information of PDB ID. Ligand information contain details such as chemical ID, molecular weight, structure ID and type of chemical. The information that is retrieved is as shown in Fig. 4.18.

4.2.11 Get Gene Ontology Function

Figure 4.19 shows the code of get gene ontology function.

Function get_gene_onto() returns gene ontology information linked to the PDB ID. The gene ontology information retrieved is shown in Fig. 4.20.

```
def get_gene_onto(pdb_id):

    out = get_info(pdb_id, url_root = 'http://www.rcsb.org/pdb/rest/goTerms?structureId=')
    out = to_dict(out)
    if not out['goTerms']:
        return None
    out = remove_at_sign(out['goTerms'])
    return out
```

Fig. 4.19 Get gene ontology function

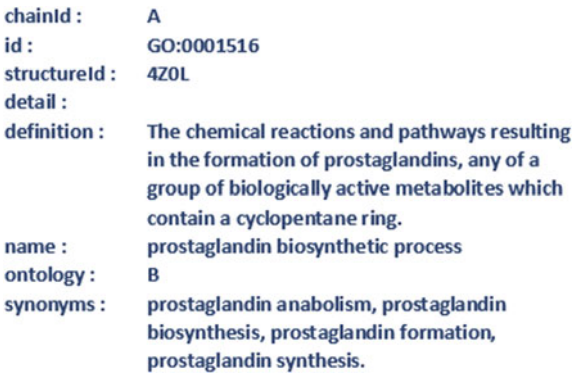


Fig. 4.20 Sample output for get gene ontology function

4.2.12 Get Sequence Cluster Function

Figure 4.21 shows the code construction of get sequence cluster function in Visual Studio.

Function get_seq_cluster() retrieves the sequence cluster of the assigned PDB ID with a character chain offset. For example, instead of a normal 4 character PDB ID, it adds a decimal behind which results in XXXX.X. An example of the sequence cluster retrieved for a PDB ID chain, 2F5N.A, is shown in Fig. 4.22.

```
def get_seq_cluster(pdb_id_chain):

    url_root = 'http://www.rcsb.org/pdb/rest/sequenceCluster?structureId='
    out = get_info(pdb_id_chain, url_root = url_root)
    out = to_dict(out)
    return remove_at_sign(out['sequenceCluster'])
```

Fig. 4.21 Get sequence cluster function

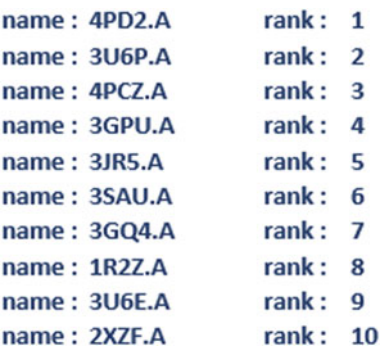


Fig. 4.22 Sample output for get sequence cluster function

4.2.13 Get Blast Function

Figure 4.23 shows the code and structure of get blast function.

The get_blast() function retrieves BLAST results for the user inputted PDB ID. The search result will return as a form of a nested dictionary which contains all the BLAST results and their metadata. For example, when an entry of 2F5N.A is entered as the PDB ID, the returned result is as shown in Fig. 4.24.

4.2.14 Get PFAM Function

Figure 4.25 shows the way get PFAM function is constructed in Visual Studio.

The get_pfam() function returns PFAM annotations for a PDB ID. The PFAM annotations result is as shown in Fig. 4.26.

```
def get_blast(pdb_id, chain_id='A'):

    raw_results = get_raw_blast(pdb_id, output_form='XML', chain_id=chain_id)

    out = xmltodict.parse(raw_results, process_namespaces=True)
    out = to_dict(out)
    out = out['BlastOutput']
    return out
```

Fig. 4.23 Get blast function

```
PELPEVETVRRELEKRIVGQKIISIEATYPRMVL--
GFEQLKKELTGKTIQGISRRGKYLIFEIGDDFRLISHLRMEGKYRLATLDAPREKHDHLTMKFADG-
QLIYADVRKFGTWELISTDQVLPYFLKKKIGPEPTYEDFDEKLFREKLRKSTKKIKPYLLEQTLVAGLGNIYVDEVLWLAK
IHPEKETNQLIESSIHLLHDSIIEILQKAIKLGGSSIRTY-
SALGSTGKMQNELQVYGKTGEKCSRCGAEIQKIKVAGRGTHFCPVCQQ
```

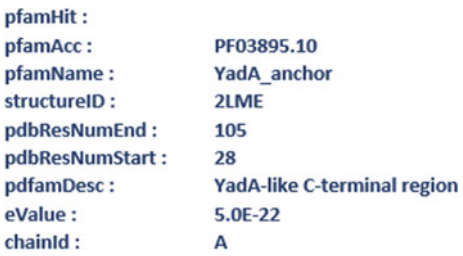
Fig. 4.24 Sample output for get blast function

```
def get_pfam(pdb_id):

    out = get_info(pdb_id, url_root = 'http://www.rcsb.org/pdb/rest/hmmer?structureId=')
    out = to_dict(out)
    if not out['hmmer3']:
        return dict()
    return remove_at_sign(out['hmmer3'])
```

Fig. 4.25 Get PFAM function

Fig. 4.26 Sample output for get PFAM function



4.2.15 Get Clusters Function

Figure 4.27 shows the code for get cluster function.

The get_clusters() function returns cluster related web services for a PDB ID. For example, the representative cluster for 4hhb.A is 2W72.A as shown in Fig. 4.28.

4.2.16 Find Results Generator Function

Figure 4.29 shows the structure and codes for find results generator function.

Function find_results_gen() outputs a generator for results returned by any search of the protein data bank conducted internally. A sample result is shown in Fig. 4.30.

4.2.17 Parse Results Generator Function

Figure 4.31 shows the code and structure for the parse results generator function.

Function parse_results_gen() queries PDB with a specific search term and field without violating the existing limitations of the API. If the search result exceeds the

```
def get_clusters(pdb_id):

    out = get_info(pdb_id, url_root = 'http://www.rcsb.org/pdb/rest/representatives?structureId=')
    out = to_dict(out)
    return remove_at_sign(out['representatives'])
```

Fig. 4.27 Get clusters function

Fig. 4.28 Sample output for get clusters function

```
def find_results_gen(search_term, field='title'):

    scan_params = make_query(search_term, querytype='AdvancedKeywordQuery')
    search_result_ids = do_search(scan_params)

    all_titles = []
    for pdb_result in search_result_ids:
        result= describe_pdb(pdb_result)
        if field in result.keys():
            yield result[field]
```

Fig. 4.29 Find results generator function

```
MYOSIN II DICTYOSTELIUM DISCOIDEUM MOTOR DOMAIN S456Y BOUND WITH MGADP-BEFX
MYOSIN II DICTYOSTELIUM DISCOIDEUM MOTOR DOMAIN S456Y BOUND WITH MGADP-ALF4
MYOSIN II DICTYOSTELIUM DISCOIDEUM MOTOR DOMAIN S456E BOUND WITH MGADP-BEFX
MYOSIN II DICTYOSTELIUM DISCOIDEUM MOTOR DOMAIN S456E BOUND WITH MGADP-ALF4
The structural basis of blebbistatin inhibition and specificity for myosin II
```

Fig. 4.30 Sample output for find results generator function

```
def parse_results_gen(search_term, field='title', max_results = 100, sleep_time=.1):

    if max_results*sleep_time > 30:
        warnings.warn("Because of API limitations, this function\
        will take at least " + str(max_results*sleep_time) + " seconds to return results.\
        If you need greater speed, try modifying the optional argument sleep_time=.1, (although \
        this may cause the search to time out)" )

    all_data_raw = find_results_gen(search_term, field=field)
    all_data =list()
    while len(all_data) < max_results:
        all_data.append(all_data_raw.send(None))
        time.sleep(sleep_time)

    return all_data
```

Fig. 4.31 Parse results generator function

limit, a warning message is displayed to the user to notify that the results are returned in a timely manner but may be incomplete.

4.2.18 Find Papers Function

Figure 4.32 shows the code for find papers function.

The function find_papers() searches the RCSB PDB for top papers according to the keyword relevancy and returns the results as a list. If the search result exceeds

```
def find_papers(search_term, **kwargs):

    all_papers = parse_results_gen(search_term, field='title', **kwargs)
    return remove_dupes(all_papers)
```

Fig. 4.32 Find papers function

```
Crystal structure of a CRISPR-associated protein from thermos thermophilus.
CRYSTAL STRUCTURE OF HYPOTHETICAL PROTEIN SS01404 FROM SULFOLOBUS SOLFATARICUS P2
NMR solution structure of a CRISPR repeat binding protein.
```

Fig. 4.33 Sample output for find papers function

the limitations of the API, an error is displayed as mentioned. As an example, the search result for the term 'crispr' is displayed in Fig. 4.33.

4.2.19 Find Authors Function

Figure 4.34 shows the constructed structure and code of the find authors function.

The purpose of the find_authors() function is the same as the find_papers function, just that it searches top authors instead. It searches based on the number of PDB entries that an author has his or her name linked with and it is not judged by the order of the author nor the ranking of the entry. Therefore, if an author has published a significant number of papers related to the search term, their work will have priority over any other author who wrote fewer papers that are most likely related to the search term used. An example is shown in Fig. 4.35 when the title 'crispr' is used as the search term.

```
def find_authors(search_term, **kwargs):

    all_individuals = parse_results_gen(search_term, field='citation_authors', **kwargs)

    full_author_list = []
    for individual in all_individuals:
        individual = individual.replace('.,', '.;')
        author_list_clean = [x.strip() for x in individual.split(';')]
        full_author_list+=author_list_clean

    out = list(chain.from_iterable(repeat(ii, c) for ii,c in Counter(full_author_list).most_common()))

    return remove_dupes(out)
```

Fig. 4.34 Find authors function

Doudna, J.A., Jinek, M., Ke, A., Li, H., Nam, K.J.

Fig. 4.35 Sample output for find authors function

4.2.20 Find Dates Function

Figure 4.36 shows find dates function structure and code.

The function find_dates() has the same usage as the 2 functions above, except that it is used to retrieve results from RCSB PDB based on the PDB submission dates. It can be utilized to retrieve data on the popularity of the given search term.

4.2.21 List Taxonomy Function

Figure 4.37 shows the code and structure built in Visual Studio for the list taxonomy function.

The list_taxa() function examines and returns any taxonomy related information provided within the description from search results that are returned by the

```
def find_dates(search_term, **kwargs):

    all_dates = parse_results_gen(search_term, field='deposition_date', **kwargs)
    return all_dates
```

Fig. 4.36 Find dates function

```
def list_taxa(pdb_list, sleep_time=.1):

    if len(pdb_list)*sleep_time > 30:
        warnings.warn("Because of API limitations, this function\
        will take at least " + str(len(pdb_list)*sleep_time) + " seconds to return results.\
        If you need greater speed, try modifying the optional argument sleep_time=.1, (although \
        this may cause the search to time out)" )

    taxa = []

    for pdb_id in pdb_list:
        all_info = get_all_info(pdb_id)
        species_results = walk_nested_dict(all_info, 'Taxonomy', maxdepth=25,outputs=[])
        first_result = walk_nested_dict(species_results,'@name',outputs=[])
        if first_result:
            taxa.append(first_result[-1])
        else:
            taxa.append('Unknown')

        time.sleep(sleep_time)

    return taxa
```

Fig. 4.37 List taxonomy function

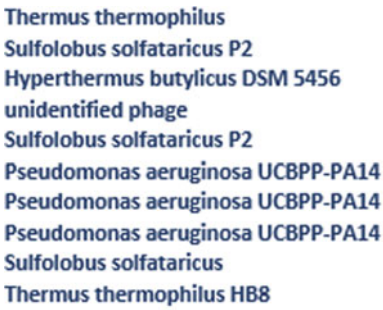


Fig. 4.38 Sample output for list taxonomy function

get_all_info() function. Descriptions from the PDB website includes the species name in each of their entries and occasionally has information of body parts or organs. For example, if the user searched for 'crispr', the result returned are as shown in Fig. 4.38.

4.2.22 List Types Function

Figure 4.39 shows the code structure of list types function.

The list_types() function analyses the list of PDB IDs provided and searches the associated structure type of PDB IDs as shown in Fig. 4.39. As an example, when a search was conducted for the keyword 'cripsr', the search result returned will show that it is categorized as a protein.

```
def list_types(pdb_list, sleep_time=.1):

    if len(pdb_list)*sleep_time > 30:
        warnings.warn("Because of API limitations, this function\
        will take at least " + str(len(pdb_list)*sleep_time) + " seconds to return results.\
        If you need greater speed, try modifying the optional argument sleep_time=.1, (although \
        this may cause the search to time out)" )

    infotypes = []
    for pdb_id in pdb_list:
        all_info = get_all_info(pdb_id)
        type_results = walk_nested_dict(all_info, '@type', maxdepth=25,outputs=[])
        if type_results:
            infotypes.append(type_results[-1])
        else:
            infotypes.append('Unknown')
        time.sleep(sleep_time)

    return infotypes
```

Fig. 4.39 List types function

4.3 Functions for Querying Information with PDB ID

4.3.1 To Dictionary Function

Figure 4.40 shows the code of to dictionary function.

The to_dict() function converts and returns a compressed form of OrderedDict(), a nested object, as a normal dictionary.

4.3.2 Remove at Sign Function

Figure 4.41 shows the code for the remove at sign function.

The remove_at_sign() function as the name suggests, removes any '@' character from the start of key names in a dictionary.

4.3.3 Remove Duplicates Function

Figure 4.42 shows the remove duplicates function code structure.

The remove_dupes() function removes any duplicated entries from the search list while not interfering with the order. The standard equivalence testing method for Python is used to find out whether there are any elements in a list that are identical to each other. For example, if there are entries of the number 1, 2, 3, 2, 4 and 5, the final appearance is shown as 1, 2, 3, 4 and 5 instead.

```
def to_dict(odict):

    out = loads(dumps(odict))
    return out
```

Fig. 4.40 To dictionary function

```
def remove_at_sign(kk):

    tagged_keys = [thing for thing in kk.keys() if thing.startswith('@')]
    for tag_key in tagged_keys:
        kk[tag_key[1:]] = kk.pop(tag_key)

    return kk
```

Fig. 4.41 Remove at sign function

```
def remove_dupes(list_with_dupes):

    visited = set()
    visited_add = visited.add
    out = [ entry for entry in list_with_dupes if not (entry in visited or visited_add(entry))]
    return out
```

Fig. 4.42 Remove duplicates function

4.3.4 Walk Nested Dictionary Function

Figure 4.43 shows the structure and code in Visual Studio for the walk nested dictionary function.

A nested dictionary may contain huge lists of other dictionaries with unknown lengths within. Therefore, a depth-first search method is used to find out whether a key is in any of the dictionaries. The maxdepth variable can be toggled to determine the maximum depth needed to search a nested dictionary for the desired result.

```
def walk_nested_dict(my_result, term, outputs=[], depth=0, maxdepth=25):

    if depth > maxdepth:
        warnings.warn('Maximum recursion depth exceeded. Returned None for the search results,'+
                      ' try increasing the maxdepth keyword argument.')
        return None


    depth = depth + 1

    if type(my_result)==dict:
        if term in my_result.keys():
            outputs.append(my_result[term])

        else:
            new_results = list(my_result.values())
            walk_nested_dict(new_results, term, outputs=outputs, depth=depth,maxdepth=maxdepth)

    elif type(my_result)==list:
        for item in my_result:
            walk_nested_dict(item, term, outputs=outputs, depth=depth,maxdepth=maxdepth)

    else:
        pass
        # dead leaf

    # this conditional may not be necessary
    if outputs:
        return outputs
    else:
        return None
```

Fig. 4.43 Walk nested dictionary function

Chapter 5
Results and Discussion

For this research, the structure of the query framework that has been explained in Chap. 4 is implemented on Microsoft Azure. The query framework can be accessed in the form of a web portal through any web browsing application, for example, Internet Explorer, Microsoft Edge, Google Chrome and others. The web portal is built to be user friendly and easy to navigate to retrieve data from RCSB PDB. The results of the query web portal are shown in this chapter.

5.1 Query Web Portal

Figure 5.1 display the homepage of the query web portal built. The web portal is built to enable users and researchers in Malaysia to be able to access the system with ease for protein ontology query purposes.

Figure 5.2 shows the search page of the query web portal. This search function enables users to search the RCSB PDB with their desired keyword. For example, a search for data relevant to 'crispr' is entered in the search field as shown above.

Figure 5.3 displays the search result for the keyword 'crispr'. As displayed in this figure, the search function works as intended. The search webpage displays all the relevant PDB ID and information for the requested search.

Figure 5.4 shows the information related to Protein ID '1WJ9'. The full information of the PDB ID obtained from the search query can be further elaborated when it is selected. As shown in Fig. 5.4, the information that can be accessed are protein description, molecule, journal, atom sequence, unit cell for cyst, unit cell for origx, unit cell for scale, helices, sequence residue and sheets.

Figure 5.5 shows the detailed information of protein ID '1WJ9'. Each of the PDB ID attributes can be further expanded through selection to display the full information for each attribute.

C. Y. Cho et al., *Large Scale Data Analytics*,
Studies in Computational Intelligence 806,
https://doi.org/10.1007/978-3-030-03892-2_5

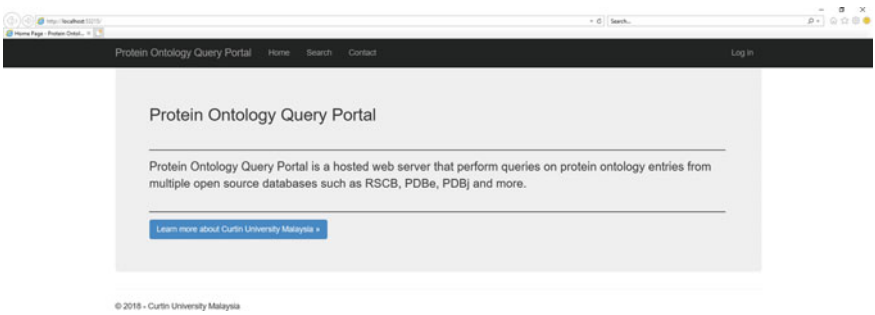


Fig. 5.1 Homepage of query web portal

Fig. 5.2 Search page of query web portal

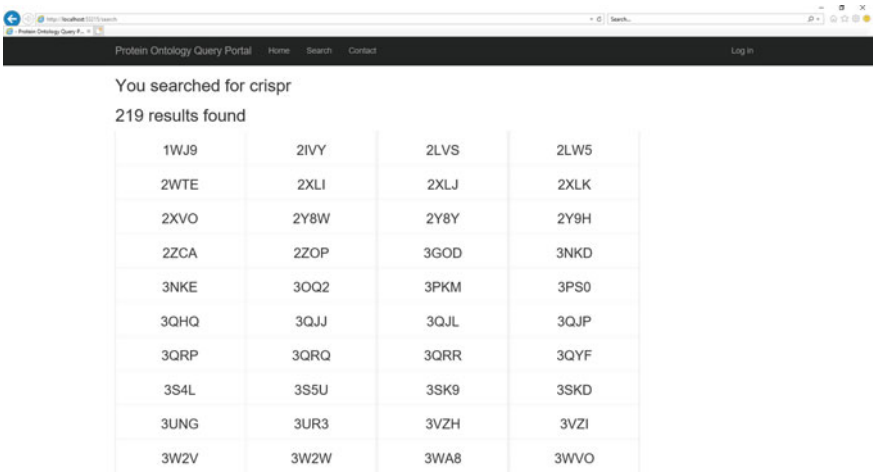


Fig. 5.3 Search result for keyword 'crispr'

Figure 5.6 shows the contact page of the query web portal. The contact information displayed on the webpage enables users or researchers to give feedback on the query web portal.

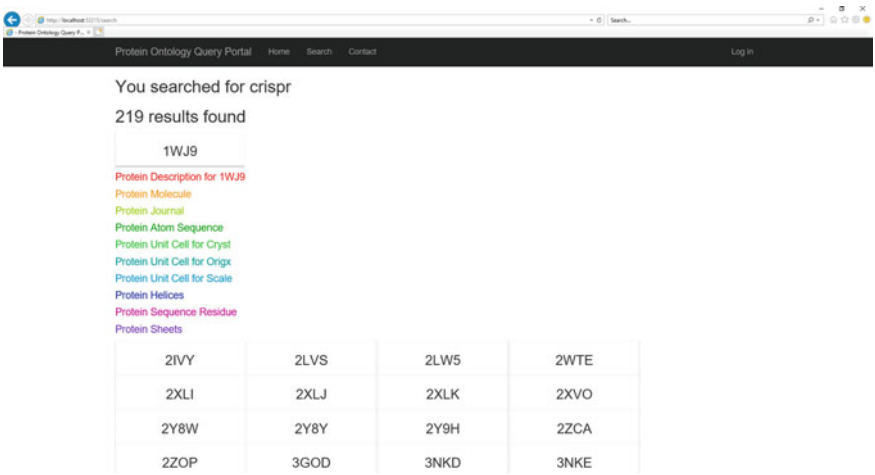


Fig. 5.4 Information related to protein ID '1WJ9'

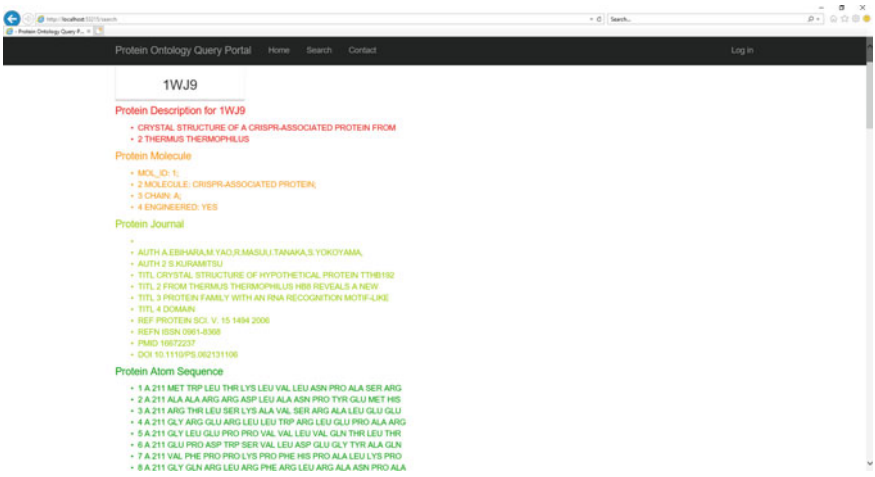


Fig. 5.5 Detailed information of protein ID '1WJ9'

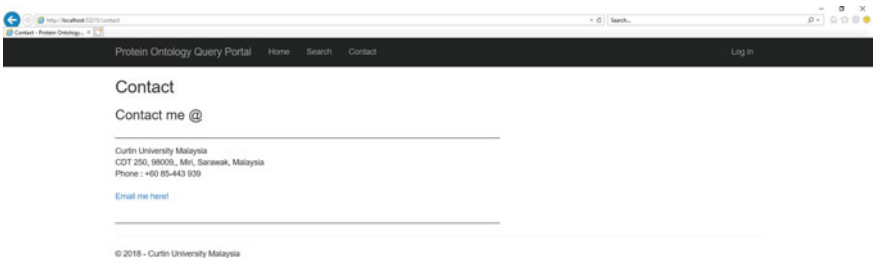


Fig. 5.6 Contact page of query web portal

5.2 Summary

In this research, we presented a query framework using Language Integrated Query and built a web portal for users to query RCSB PDB. The query framework presented is built based on the Language Integrated Query with python language. Results show that the query framework is capable of querying and providing users with their desired results. To provide sufficient computing resources for the query framework, it is deployed on a cloud computing platform, Microsoft Azure. This enables the framework to query without facing issues involving insufficient resources that may cause the framework to crash. There are certain limitations that are limiting the performance of this framework and these limitations will be discussed in Chap. 6.

Chapter 6
Conclusion and Future Works

6.1 Conclusion

The study of this research shows the difficulties faced by the current generation for database querying. Recent methodologies such as semantic integration focuses on data integration, data mapping and data translation. These approaches can be done for small to medium data sources. However, when it comes to querying databases that are huge and are being constantly updated by users around the world, these approaches are not suitable and not cost effective.

To overcome these challenges from a different perspective, we presented a different querying method using Language Integrated Query in this research. Instead of integrating existing datasets from different data sources into a single source, we used Language Integrated Query to build a query framework that is capable of querying directly from sources without the need for data translation or integration. To ensure that there are no performance issues, the query framework is implemented on a cloud computing environment, Microsoft Azure, to utilize the vast computing resources available there. A user-friendly web portal was built and implemented on Microsoft Azure for users to search and query the RCSB PDB without any issue.

Through the construction and implementation of the query framework, the framework can perform thorough searches through RCSB PDB for results as planned. The search might take a longer period to be performed depending on the keyword or query that has been searched or requested by the user due to certain limitations on both client and server side. There are several limitations and these are discussed in the next section of this chapter.

C. Y. Cho et al., *Large Scale Data Analytics*,
Studies in Computational Intelligence 806,
https://doi.org/10.1007/978-3-030-03892-2_6

6.2 Limitations

There are several factors that limit the capabilities of the query framework to function smoothly with minimal delays.

These issues can be improved through several methods:

1. Upgrading of the existing RCSB PDB server infrastructure, mainly hardware, connection and software wise.
2. Increasing the resources of Microsoft Azure virtual machine, resulting in an increase in expenses to maintain existing cloud computing infrastructure of Curtin University Malaysia.
3. Changing the hosting location of virtual machine to the nearest hosting site for RCSB PDB, in this case, United States of America.

However, the main issue that has been presented is with the technology we currently have, it is still difficult to solve the issue of hosting large scale data and ensuring all operations run smoothly. Due to the large number of researchers and users using RCSB PDB, it is hard for the RCSB PDB server to cater to the needs of all these requests without having a latency issue. Therefore, the delay in querying RCSB PDB is due to the latency issue and the hardware limitation issue.

Hardware on the web portal deployment plays a huge part in this as well. If the hardware performance is insufficient, the framework will crash depending on the number of queries and users.

6.3 Future Works

For future development, the infrastructure hosting the Microsoft Azure cloud computing platform can be improved and improvised to withstand the stress imposed by the query framework on the hardware available under heavy usage. However, this method will increase the cost of the project.

Other than that, the program can be further optimized to decrease the latency and stress load imposed on the hosting server. The existing search functions in the program can be fashioned into an advanced search that can be featured in the web portal as well as to only search and return a very specific component of a protein data from RCSB PDB.

Appendix

(A) **Query Codes**

C. Y. Cho et al., *Large Scale Data Analytics*,
Studies in Computational Intelligence 806,
https://doi.org/10.1007/978-3-030-03892-2

```
"""

Definition of views.

"""
from django.shortcuts import render
from django.http import HttpRequest, HttpResponse
from django.template import RequestContext
from datetime import datetime
from django import *
import django.middleware.csrf
from app.forms import *
import json
from django.template.loader import render_to_string
import pprint
r_dict={'name':'value', 'description':'value','molecule':'value','journal':'value' }

def set_default(obj):
    if isinstance(obj, set):
        return list(obj)
    raise TypeError

def home(request):
    """Renders the home page."""
    assert isinstance(request, HttpRequest)
    return render(
        request,
        'app/index.html',
        {
            'title':'Home Page',
            'year':datetime.now().year,
        }
    )

def contact(request):
    """Renders the contact page."""
    assert isinstance(request, HttpRequest)
    return render(
        request,
        'app/contact.html',
        {
            'title':'Contact',
            'message':'Contact me @',
            'year':datetime.now().year,
        }
    )
```

```
def search(request):
    """Renders the about page."""
    assert isinstance(request, HttpRequest)
    r_dict={}
    if request.method== 'GET':
        return render(
            request,
            'app/about.html',
            {
                'title':'Search',
                'message':'Key in your search term.',
                'year':datetime.now().year,
                'current_name':'actin network',
            }
        )
    elif request.is_ajax():
        #check fr results
        data =request.POST.get("postDiv")
        protein =request.POST.get("protein")
        b=get_pdb_file(protein)
        r_dict['description']=get_description(b)
        r_dict['molecule']=get_molecule(b)
        r_dict['journal']=get_journal(b)
        r_dict['atom']=get_atomSeq(b)
        r_dict['uc_cryst']=get_unitCell_cryst(b)
        r_dict['uc_origx']=get_unitCell_origx(b)
        r_dict['uc_scale']=get_unitCell_scale(b)
        r_dict['helices']=get_helices(b)
        r_dict['sheets']=get_sheets(b)
        r_dict['seqres']=get_seqRes(b)

        return
HttpResponse(json.dumps({'array':r_dict,'test':r_dict['sheets']}),content_type="appli
cation/json")

    else:
        key=request.POST.get('your_name')

        search_dict = make_query(key)
        found_pdbs = do_search(search_dict)
        found_pdbs1 = [str(x) for x in found_pdbs]
        list=[]
        onlyprotein=[]
        for i in found_pdbs1:
            r_dict['name']=i
            onlyprotein.append(i)
        return render(request,
                    'app/premature.html',
                    {
                        'data':list,
```

```
                'searchTerm':key,
                'size':len(onlyprotein),
                'onlyprotein':onlyprotein,
            }
        )

def r_query(term,request):
    search_dict = make_query('actin network')
    found_pdbs = do_search(search_dict)
    found_pdbs1 = [str(x) for x in found_pdbs]
    formatted_data = simpletable.fit_data_to_columns(found_pdbs1,5)
    table = simpletable.SimpleTable(formatted_data)
    html_page = simpletable.HTMLPage(table)
    html_page.save("app\premature.html")
    return render(
        request,
        'app/premature.html',
        {
            'title':'Search',
            'message':'Key in your search term.',
            'year':datetime.now().year,
            'current_name':' E.G. actin network',
        }
    )

def oriresult(key,request):
        return render(
        request,
        'app/index.html',
        {
            'title':'Result',
            'message':key,
            'year':datetime.now().year,

        }
        )

def resulttest(request):
    """Renders the contact page."""
    assert isinstance(request, HttpRequest)
    if request.is_ajax():
        data =request.POST.get("postDiv")
```

```
    return HttpResponse(json.dumps({'name':data}),content_type="application/json")
    else:
        return render(
            request,
            'app/searchresult.html',
            {
                'title':'Contact',
                'message':'Contact me @',
                'year':datetime.now().year,
                'csrf_token':django.middleware.csrf.get_token(request)

            }

    )
def reroute(request):
    """Renders the contact page."""
    assert isinstance(request, HttpRequest)
    return render(
        request,
        'app/reroute.html',
        {
            'title':'Contact',
            'message':'Contact me @',
            'year':datetime.now().year,
        }
    )

##Entry
def get_description(string):
    pattern='.*TITLE(.*)[;\n]'
    description=re.findall(pattern, string)#, re.M|re.I
    return description

def get_molecule(string):
    pattern='.*COMPND(.*)[;\n]'
    molecule=re.findall(pattern, string)
    return molecule

def get_journal(string):
    pattern='.*JRNL(\s{3}.*)[;\n]'
    journal=re.findall(pattern, string)
    return journal

##Structure
def get_atomSeq(string):
    pattern='.*SEQRES(.*)[;\n]'
    atomSeq=re.findall(pattern, string)
    return atomSeq

def get_unitCell_cryst(string):
```

```
    pattern='CRYST(\d+\s{3}.*)\n'
    cryst=re.findall(pattern, string)
    return cryst

def get_unitCell_origx(string):
    pattern='ORIGX(\d+\s+.*)\n'
    origx=re.findall(pattern, string)
    return origx

def get_unitCell_scale(string):
    pattern='SCALE(\d+\s+.*)\n'
    scale=re.findall(pattern, string)
    return scale#.group(1)

#structuralDomain
def get_helices(string):
    pattern='HELIX(\s+.*)\n'
    helices=re.findall(pattern, string)
    return helices

def get_sheets(string):
    pattern='SHEET(\s+.*)\n'
    sheets=re.findall(pattern, string)
    return sheets

def get_seqRes(string):
    pattern='SEQRES(\s+.*)\n'
    seqRes=re.findall(pattern, string)
    return seqRes

'''
PyPDB: A Python API for the RCSB Protein Data Bank
'''
from collections import OrderedDict, Counter
from itertools import repeat, chain
import urllib.request
import time
import re
from json import loads, dumps
import warnings

import xmltodict

try:
    from bs4 import BeautifulSoup
except ImportError:
    try:
        import BeautifulSoup
    except ImportError:
```

```
print ("pypdb can't find BeautifulSoup. You cannot parse BLAST search results
without this module")

'''
==================
Functions for searching the RCSB PDB for lists of PDB IDs
==================
'''

# functions for conducting searches and obtaining lists of PDB ids
def make_query(search_term, querytype='AdvancedKeywordQuery'):
    ''' Repackage strings into a search dictionary

    This function takes a list of search terms and specifications
    and repackages it as a dictionary object that can be used to conduct a search

    Parameters
    ----------
    search_term : str

        The specific term to search in the database. For specific query types,
        the strings that will yield valid results are limited to:

        'HoldingsQuery' : A Ggeneral search of the metadata associated with PDB IDs

        'ExpTypeQuery' : Experimental Method such as 'X-RAY', 'SOLID-STATE
NMR', etc

        'AdvancedKeywordQuery' : Any string that appears in the title or abstract

        'StructureIdQuery' :  Perform a search for a specific Structure ID

        'ModifiedStructuresQuery' : Search for related structures

        'AdvancedAuthorQuery' : Search by the names of authors associated with entries

        'MotifQuery' : Search for a specific motif

        'NoLigandQuery' : Find full list of PDB IDs without free ligrands

    querytype : str

        The type of query to perform, the easiest is an AdvancedKeywordQuery but more
        specific types of searches may also be performed

    Returns
    -------

    scan_params : dict
```

A dictionary representing the query

Examples

This method usually gets used in tandem with do_search

```
>>> a = make_query('actin network')
>>> print (a)
{'orgPdbQuery': {'description': 'Text Search for: actin',
'keywords': 'actin',
'queryType': 'AdvancedKeywordQuery'}}


>>> search_dict = make_query('actin network')
>>> found_pdbs = do_search(search_dict)
>>> print(found_pdbs)
['1D7M', '3W3D', '4A7H', '4A7L', '4A7N']

>>> search_dict = make_query('T[AG]AGGY',querytype='MotifQuery')
>>> found_pdbs = do_search(search_dict)
>>> print(found_pdbs)
['3LEZ', '3SGH', '4F47']

'''
assert querytype in {'HoldingsQuery', 'ExpTypeQuery',
                     'AdvancedKeywordQuery','StructureIdQuery',
                     'ModifiedStructuresQuery', 'AdvancedAuthorQuery', 'MotifQuery',
                     'NoLigandQuery'
                    }

query_params = dict()
query_params['queryType'] = querytype

if querytype=='AdvancedKeywordQuery':
   query_params['description'] = 'Text Search for: '+ search_term
   query_params['keywords'] = search_term

elif querytype=='NoLigandQuery':
   query_params['haveLigands'] = 'yes'

elif querytype=='AdvancedAuthorQuery':
   query_params['description'] = 'Author Name: '+ search_term
   query_params['searchType'] = 'All Authors'
   query_params['audit_author.name'] = search_term
   query_params['exactMatch'] = 'false'

elif querytype=='MotifQuery':
   query_params['description'] = 'Motif Query For: '+ search_term
   query_params['motif'] = search_term
```

```
    # search for a specific structure
    elif querytype in ['StructureIdQuery','ModifiedStructuresQuery']:
        query_params['structureIdList'] = search_term


    elif querytype=='ExpTypeQuery':
        query_params['experimentalMethod'] = search_term
        query_params['description'] = 'Experimental Method Search : Experimental
Method='+ search_term
        query_params['mvStructure.expMethod.value']= search_term


    scan_params = dict()
    scan_params['orgPdbQuery'] = query_params


    return scan_params

    def do_search(scan_params):
    '''Convert dict() to XML object an then send query to the RCSB PDB

    This function takes a valid query dict() object, converts it to XML,
    and then sends a request to the PDB for a list of IDs corresponding to search results

    Parameters
    ----------

    scan_params : dict
        A dictionary of query attributes to use for
        the search of the PDB

    Returns
    -------

    idlist : list
        A list of PDB ids returned by the search

    Examples
    --------
    This method usually gets used in tandem with make_query

    >>> a = make_query('actin network')
    >>> print (a)
    {'orgPdbQuery': {'description': 'Text Search for: actin',
    'keywords': 'actin',
    'queryType': 'AdvancedKeywordQuery'}}


    >>> search_dict = make_query('actin network')
    >>> found_pdbs = do_search(search_dict)
```

```
>>> print(found_pdbs)
['1D7M', '3W3D', '4A7H', '4A7L', '4A7N']

>>> search_dict = make_query('T[AG]AGGY',querytype='MotifQuery')
>>> found_pdbs = do_search(search_dict)
>>> print(found_pdbs)
['3LEZ', '3SGH', '4F47']
'''

url = 'http://www.rcsb.org/pdb/rest/search'

queryText = xmltodict.unparse(scan_params, pretty=False)
queryText = queryText.encode()

req = urllib.request.Request(url, data=queryText)
f = urllib.request.urlopen(req)
result = f.read()

if not result:
   warnings.warn('No results were obtained for this search')

idlist = str(result)
idlist =idlist.split('\\n')
idlist[0] = idlist[0][-4:]
kk = idlist.pop(-1)

return idlist

ef do_protsym_search(point_group, min_rmsd=0.0, max_rmsd=7.0):
'''Performs a protein symmetry search of the PDB

This function can search the Protein Data Bank based on how closely entries
match the user-specified symmetry group

Parameters
----------

point_group : str
   The name of the symmetry point group to search. This includes all the standard
   abbreviations for symmetry point groups (e.g., C1, C2, D2, T, O, I, H, A1)

min_rmsd : float
   The smallest allowed total deviation (in Angstroms) for a result to be classified
   as having a matching symmetry

max_rmsd : float
   The largest allowed total deviation (in Angstroms) for a result to be classified
   as having a matching symmetry
```

```
    Returns
    -------

    idlist : list of strings
       A list of PDB IDs resulting from the search

    Examples
    --------

    >>> kk = do_protsym_search('C9', min_rmsd=0.0, max_rmsd=1.0)
    >>> print(kk[:5])
    ['1KZU', '1NKZ', '2FKW', '3B8M', '3B8N']

    '''

    query_params = dict()
    query_params['queryType'] = 'PointGroupQuery'
    query_params['rMSDComparator'] = 'between'

    query_params['pointGroup'] = point_group
    query_params['rMSDMin'] = min_rmsd
    query_params['rMSDMax'] = max_rmsd

    scan_params = dict()
    scan_params['orgPdbQuery'] = query_params
    idlist =  do_search(scan_params)
    return idlist


ef get_all():
    """Return a list of all PDB entries currently in the RCSB Protein Data Bank

    Returns
    -------

    out : list of str
       A list of all of the PDB IDs currently in the RCSB PDB

    Examples
    --------

    >>> print(get_all()[:10])
    ['100D', '101D', '101M', '102D', '102L', '102M', '103D', '103L', '103M', '104D']

    """

    url = 'http://www.rcsb.org/pdb/rest/getCurrent'

    req = urllib.request.Request(url)
```

```
    f = urllib.request.urlopen(req)
    result = f.read()
    assert result

    kk = str(result)

    p = re.compile('structureId=\".....')
    matches = p.findall(str(result))
    out = list()
    for item in matches:
        out.append(item[-5:-1])

    return out

'''
==================
Functions for looking up information given PDB ID
==================
'''

def get_info(pdb_id,
url_root='http://www.rcsb.org/pdb/rest/describeMol?structureId='):
    '''Look up all information about a given PDB ID

    Parameters
    ----------

    pdb_id : string
        A 4 character string giving a pdb entry of interest

    url_root : string
        The string root of the specific url for the request type

    Returns
    -------

    out : OrderedDict
        An ordered dictionary object corresponding to bare xml

    '''

    url = url_root + pdb_id
    req = urllib.request.Request(url)
    f = urllib.request.urlopen(req)
    result = f.read()
    assert result

    out = xmltodict.parse(result,process_namespaces=True)

    return out
```

```
def get_pdb_file(pdb_id, filetype='pdb', compression=False):
    '''Get the full PDB file associated with a PDB_ID

    Parameters
    ----------

    pdb_id : string
        A 4 character string giving a pdb entry of interest

    filetype: string
        The file type.
        'pdb' is the older file format,
        'cif' is the newer replacement.
        'xml' an also be obtained and parsed using the various xml tools included in
PyPDB
        'structfact' is a test file containing structure information for certain entries

    compression : bool
        Retrieve a compressed (gz) version of the file

    Returns
    -------

    result : string
        The string representing the full PDB file in the given format


http://www.rcsb.org/pdb/download/downloadFile.do?fileFormat=structfact&structur
eId=2F5N

    Examples
    --------
    >>> pdb_file = get_pdb_file('4lza', filetype='cif', compression=True)
    >>> print(pdb_file[:200])
    data_4LZA
    #
    _entry.id   4LZA
    #
    _audit_conform.dict_name       mmcif_pdbx.dic
    _audit_conform.dict_version    4.032
    _audit_conform.dict_location
http://mmcif.pdb.org/dictionaries/ascii/mmcif_pdbx

    DEV NOTE:
    http://www.rcsb.org/pdb/files/2F5N.pdb1.gz

    '''

    fullurl = 'http://www.rcsb.org/pdb/download/downloadFile.do?fileFormat='
    fullurl += filetype
```

```
    if compression:
        fullurl += '&compression=YES'
    else:
        fullurl += '&compression=NO'

    fullurl += '&structureId=' + pdb_id
    # url = 'http://www.rcsb.org/pdb/files/'+pdb_id+'.pdb'
    req = urllib.request.Request(fullurl)
    f = urllib.request.urlopen(req)
    result = f.read()
    result = result.decode('unicode_escape')

    return result


def get_all_info(pdb_id):
    '''A wrapper for get_info that cleans up the output slighly

    Parameters
    ----------

    pdb_id : string
        A 4 character string giving a pdb entry of interest

    Returns
    -------

    out : dict
        A dictionary containing all the information stored in the entry

    Examples
    --------

    >>> all_info = get_all_info('4lza')
    >>> print(all_info)
    {'polymer': {'macroMolecule': {'@name': 'Adenine phosphoribosyltransferase', '
    accession': {'@id': 'B0K969'}}, '@entityNr': '1', '@type': 'protein',
    'polymerDescription': {'@description': 'Adenine phosphoribosyltransferase'},
    'synonym': {'@name': 'APRT'}, '@length': '195', 'enzClass': {'@ec': '2.4.2.7'},
    'chain': [{'@id': 'A'}, {'@id': 'B'}],
    'Taxonomy': {'@name': 'Thermoanaerobacter pseudethanolicus ATCC 33223',
    '@id': '340099'}, '@weight': '22023.9'}, 'id': '4LZA'}

    >>> results = get_all_info('2F5N')
    >>> first_polymer = results['polymer'][0]
    >>> first_polymer['polymerDescription']
    {'@description':                                                    "5'-
D(*AP*GP*GP*TP*AP*GP*AP*CP*CP*TP*GP*GP*AP*CP*GP*C)-3'"}

    '''
```

```
    out = to_dict( get_info(pdb_id) )['molDescription']['structureId']
    out = remove_at_sign(out)
    return out

def get_raw_blast(pdb_id, output_form='HTML', chain_id='A'):
    '''Look up full BLAST page for a given PDB ID

    get_blast() uses this function internally

    Parameters
    ----------

    pdb_id : string
        A 4 character string giving a pdb entry of interest

    chain_id : string
        A single character designating the chain ID of interest


    output_form : string
        TXT, HTML, or XML formatting of the outputs

    Returns
    -------

    out : OrderedDict
        An ordered dictionary object corresponding to bare xml

    '''

    url_root = 'http://www.rcsb.org/pdb/rest/getBlastPDB2?structureId='
    url = url_root + pdb_id + '&chainId='+ chain_id +'&outputFormat=' + output_form
    req = urllib.request.Request(url)
    f = urllib.request.urlopen(req)
    result = f.read()
    result = result.decode('unicode_escape')
    assert result

    return result


def parse_blast(blast_string):
    '''Clean up HTML BLAST results

    This function requires BeautifulSoup and the re module
    It goes throught the complicated output returned by the BLAST
    search and provides a list of matches, as well as the raw
    text file showing the alignments for each of the matches.

    This function works best with HTML formatted Inputs
```

```
    ------

    get_blast() uses this function internally

    Parameters
    ----------

    blast_string : str
       A complete webpage of standard BLAST results

    Returns
    -------

    out : 2-tuple
       A tuple consisting of a list of PDB matches, and a list
       of their alignment text files (unformatted)


    '''

    soup = BeautifulSoup(str(blast_string), "html.parser")

    all_blasts = list()
    all_blast_ids = list()

    pattern = '></a>....:'
    prog = re.compile(pattern)

    for item in soup.find_all('pre'):
       if len(item.find_all('a'))==1:
          all_blasts.append(item)
          blast_id = re.findall(pattern, str(item) )[0][-5:-1]
          all_blast_ids.append(blast_id)

    out = (all_blast_ids, all_blasts)
    return out


def get_blast2(pdb_id, chain_id='A', output_form='HTML'):
    '''Alternative way to look up BLAST for a given PDB ID. This function is a wrapper
    for get_raw_blast and parse_blast

    Parameters
    ----------
    pdb_id : string
       A 4 character string giving a pdb entry of interest

    chain_id : string
       A single character designating the chain ID of interest
```

```
    output_form : string
       TXT, HTML, or XML formatting of the BLAST page

    Returns
    -------

    out : 2-tuple
       A tuple consisting of a list of PDB matches, and a list
       of their alignment text files (unformatted)


    Examples
    --------

    >>> blast_results = get_blast2('2F5N', chain_id='A', output_form='HTML')
    >>> print('Total Results: ' + str(len(blast_results[0])) +'\n')
    >>> print(blast_results[1][0])
    Total Results: 84
    <pre>
    &gt;<a name="45354"></a>2F5P:3:A|pdbid|entity|chain(s)|sequence
           Length = 274
     Score =  545 bits (1404), Expect = e-155,   Method: Composition-based stats.
     Identities = 274/274 (100%), Positives = 274/274 (100%)
    Query: 1
MPELPEVETIRRTLLPLIVGKTIEDVRIFWPNIIRHPRDSEAFAARMIGQTVRG
LERRGK 60

MPELPEVETIRRTLLPLIVGKTIEDVRIFWPNIIRHPRDSEAFAARMIGQTVRG
LERRGK
    Sbjct: 1
MPELPEVETIRRTLLPLIVGKTIEDVRIFWPNIIRHPRDSEAFAARMIGQTVRG
LERRGK 60
    ...

    '''

    raw_results          =          get_raw_blast(pdb_id,          chain_id=chain_id,
output_form=output_form)
    out = parse_blast(raw_results)

    return out

def describe_pdb(pdb_id):
    """Get description and metadata of a PDB entry

    Parameters
    ----------

    pdb_id : string
       A 4 character string giving a pdb entry of interest
```

```
    Returns
    -------
    out : string
        A text pdb description from PDB

    Examples
    --------

    >>> describe_pdb('4lza')
    {'citation_authors': 'Malashkevich, V.N., Bhosle, R., Toro, R., Hillerich, B., Gizzi,
A., Garforth, S., Kar, A., Chan, M.K., Lafluer, J., Patel, H., Matikainen, B., Chamala,
S., Lim, S., Celikgil, A., Villegas, G., Evans, B., Love, J., Fiser, A., Khafizov, K.,
Seidel, R., Bonanno, J.B., Almo, S.C.',
     'deposition_date': '2013-07-31',
     'expMethod': 'X-RAY DIFFRACTION',
     'keywords': 'TRANSFERASE',
     'last_modification_date': '2013-08-14',
     'nr_atoms': '0',
     'nr_entities': '1',
     'nr_residues': '390',
     'release_date': '2013-08-14',
     'resolution': '1.84',
     'status': 'CURRENT',
     'structureId': '4LZA',
     'structure_authors': 'Malashkevich, V.N., Bhosle, R., Toro, R., Hillerich, B., Gizzi,
A., Garforth, S., Kar, A., Chan, M.K., Lafluer, J., Patel, H., Matikainen, B., Chamala,
S., Lim, S., Celikgil, A., Villegas, G., Evans, B., Love, J., Fiser, A., Khafizov, K.,
Seidel, R., Bonanno, J.B., Almo, S.C., New York Structural Genomics Research
Consortium (NYSGRC)',
     'title': 'Crystal structure of adenine phosphoribosyltransferase from
Thermoanaerobacter pseudethanolicus ATCC 33223, NYSGRC Target 029700.'}

    """
    out = get_info(pdb_id, url_root =
'http://www.rcsb.org/pdb/rest/describePDB?structureId=')
    out = to_dict(out)
    out = remove_at_sign(out['PDBdescription']['PDB'])
    return out

def get_entity_info(pdb_id):
    """Return pdb id information

    Parameters
    ----------

    pdb_id : string
        A 4 character string giving a pdb entry of interest

    Returns
    -------
```

```
    out : dict
       A dictionary containing a description the entry

    Examples
    --------
    >>> get_entity_info('4lza')
       {'Entity': {'@id': '1',
     '@type': 'protein',
     'Chain': [{'@id': 'A'}, {'@id': 'B'}]},
     'Method': {'@name': 'xray'},
     'bioAssemblies': '1',
     'release_date': 'Wed Aug 14 00:00:00 PDT 2013',
     'resolution': '1.84',
     'structureId': '4lza'}

    """
    out = get_info(pdb_id, url_root =
'http://www.rcsb.org/pdb/rest/getEntityInfo?structureId=')
    out = to_dict(out)
    return remove_at_sign( out['entityInfo']['PDB'] )

def describe_chemical(chem_id):
    """

    Parameters
    ----------

    chem_id : string
       A 4 character string representing the full chemical sequence of interest (ie, NAG)

    Returns
    -------

    out : dict
       A dictionary containing the chemical description associated with the PDB ID

    Examples
    --------
    >>> chem_desc = describe_chemical('NAG')
    >>> print(chem_desc)
    {'describeHet': {'ligandInfo': {'ligand': {'@molecularWeight': '221.208',
    'InChIKey': 'OVRNDRQMDRJTHS-FMDGEEDCSA-N', '@type': 'D-saccharide',
    'chemicalName': 'N-ACETYL-D-GLUCOSAMINE', '@chemicalID': 'NAG',
    'smiles': 'CC(=O)N[C@@H]1[C@H]([C@@H]([C@H](O[C@H]1O)CO)O)O', '
    InChI': 'InChI=1S/C8H15NO6/c1-3(11)9-5-7(13)6(12)4(2-10)15-8(5)14/
    h4-8,10,12-14H,2H2,1H3,(H,9,11)/t4-,5-,6-,7-,8-/m1/s1',
    'formula': 'C8 H15 N O6'}}}}

    """
```

```
    out = get_info(chem_id, url_root =
'http://www.rcsb.org/pdb/rest/describeHet?chemicalID=')
    out = to_dict(out)
    return out

def get_ligands(pdb_id):
    """Return ligands of given PDB ID

    Parameters
    ----------

    pdb_id : string
        A 4 character string giving a pdb entry of interest

    Returns
    -------

    out : dict
        A dictionary containing a list of ligands associated with the entry


    Examples
    --------
    >>> ligand_dict = get_ligands('100D')
    >>> print(ligand_dict)
    {'id': '100D',
    'ligandInfo': {'ligand': {'@chemicalID': 'SPM',
                    '@molecularWeight': '202.34',
                    '@structureId': '100D',
                    '@type': 'non-polymer',
                    'InChI':      'InChI=1S/C10H26N4/c11-5-3-9-13-7-1-2-8-14-10-4-6-
12/h13-14H,1-12H2',
                    'InChIKey': 'PFNFFQXMRSDOHW-UHFFFAOYSA-N',
                    'chemicalName': 'SPERMINE',
                    'formula': 'C10 H26 N4',
                    'smiles': 'C(CCNCCCN)CNCCCN'}}}

    """
    out                =                get_info(pdb_id,                url_root                =
'http://www.rcsb.org/pdb/rest/ligandInfo?structureId=')
    out = to_dict(out)
    return remove_at_sign(out['structureId'])

def get_gene_onto(pdb_id):
    """Return ligands of given PDB_ID

    Parameters
    ----------

    pdb_id : string
```

```
    A 4 character string giving a pdb entry of interest

    Returns
    -------

    out : dict
       A dictionary containing the gene ontology information associated with the entry

    Examples
    --------

    >>> gene_info = get_gene_onto('4Z0L')
    >>> print(gene_info['term'][0])
    {'@chainId': 'A',
     '@id': 'GO:0001516',
     '@structureId': '4Z0L',
     'detail': {'@definition': 'The chemical reactions and pathways resulting '
                               'in the formation of prostaglandins, any of a '
                               'group of biologically active metabolites which '
                               'contain a cyclopentane ring.',
                '@name': 'prostaglandin biosynthetic process',
                '@ontology': 'B',
                '@synonyms': 'prostaglandin anabolism, prostaglandin '
                             'biosynthesis, prostaglandin formation, '
                             'prostaglandin synthesis'}}
    """
    out = get_info(pdb_id, url_root =
'http://www.rcsb.org/pdb/rest/goTerms?structureId=')
    out = to_dict(out)
    if not out['goTerms']:
        return None
    out = remove_at_sign(out['goTerms'])
    return out

def get_seq_cluster(pdb_id_chain):
    """Get the sequence cluster of a PDB ID plus a pdb_id plus a chain,

    Parameters
    ----------

    pdb_id_chain : string
       A string denoting a 4 character PDB ID plus a one character chain
       offset with a dot: XXXX.X, as in 2F5N.A

    Returns
    -------

    out : dict
       A dictionary containing the sequence cluster associated with the PDB
       entry and chain
```

```
    Examples
    --------

    >>> sclust = get_seq_cluster('2F5N.A')
    >>> print(sclust['pdbChain'][:10])
    [{'@name': '4PD2.A', '@rank': '1'},
     {'@name': '3U6P.A', '@rank': '2'},
     {'@name': '4PCZ.A', '@rank': '3'},
     {'@name': '3GPU.A', '@rank': '4'},
     {'@name': '3JR5.A', '@rank': '5'},
     {'@name': '3SAU.A', '@rank': '6'},
     {'@name': '3GQ4.A', '@rank': '7'},
     {'@name': '1R2Z.A', '@rank': '8'},
     {'@name': '3U6E.A', '@rank': '9'},
     {'@name': '2XZF.A', '@rank': '10'}]

    """

    url_root = 'http://www.rcsb.org/pdb/rest/sequenceCluster?structureId='
    out = get_info(pdb_id_chain, url_root = url_root)
    out = to_dict(out)
    return remove_at_sign(out['sequenceCluster'])

def get_blast(pdb_id, chain_id='A'):
    """
    Return BLAST search results for a given PDB ID
    The key of the output dict())that outputs the full search results is
    'BlastOutput_iterations'

    To get a list of just the results without the metadata of the search use:
    hits = full_results['BlastOutput_iterations']['Iteration']['Iteration_hits']['Hit']

    Parameters
    ----------
    pdb_id : string
        A 4 character string giving a pdb entry of interest

    chain_id : string
        A single character designating the chain ID of interest


    Returns
    -------

    out : dict()
        A nested dict() consisting of the BLAST search results and all associated
metadata
        If you just want the hits, look under four levels of keys:
        results['BlastOutput_iterations']['Iteration']['Iteration_hits']['Hit']
```

```
    Examples
    --------

    >>> blast_results = get_blast('2F5N', chain_id='A')
    >>> just_hits =
blast_results['BlastOutput_iterations']['Iteration']['Iteration_hits']['Hit']
    >>> print(just_hits[50]['Hit_hsps']['Hsp']['Hsp_hseq'])
    PELPEVETVRRELEKRIVGQKIISIEATYPRMVL--
TGFEQLKKELTGKTIQGISRRGKYLIFEIGDDFRLISHLRMEGKYRLATLDAP
REKHDHL
    TMKFADG-
QLIYADVRKFGTWELISTDQVLPYFLKKKIGPEPTYEDFDEKLFREKLRKSTK
KIKPYLLEQTLVAGLGNIYVDEVLWLAKIHPEKET
    NQLIESSIHLLHDSIIEILQKAIKLGGSSIRTY-
SALGSTGKMQNELQVYGKTGEKCSRCGAEIQKIKVAGRGTHFCPVCQQ


    """

    raw_results = get_raw_blast(pdb_id, output_form='XML', chain_id=chain_id)

    out = xmltodict.parse(raw_results, process_namespaces=True)
    out = to_dict(out)
    out = out['BlastOutput']
    return out


def get_pfam(pdb_id):
    """Return PFAM annotations of given PDB_ID

    Parameters
    ----------

    pdb_id : string
        A 4 character string giving a pdb entry of interest

    Returns
    -------

    out : dict
        A dictionary containing the PFAM annotations for the specified PDB ID

    Examples
    --------

    >>> pfam_info = get_pfam('2LME')
    >>> print(pfam_info)
    {'pfamHit': {'@pfamAcc': 'PF03895.10', '@pfamName': 'YadA_anchor',
    '@structureId': '2LME', '@pdbResNumEnd': '105', '@pdbResNumStart': '28',
```

```
'@pfamDesc': 'YadA-like C-terminal region', '@eValue': '5.0E-22', '@chainId': 'A'}}

    """
    out = get_info(pdb_id, url_root =
'http://www.rcsb.org/pdb/rest/hmmer?structureId=')
    out = to_dict(out)
    if not out['hmmer3']:
        return dict()
    return remove_at_sign(out['hmmer3'])

def get_clusters(pdb_id):
    """Return cluster related web services of given PDB_ID

    Parameters
    ----------

    pdb_id : string
        A 4 character string giving a pdb entry of interest

    Returns
    -------

    out : dict
        A dictionary containing the representative clusters for the specified PDB ID

    Examples
    --------

    >>> clusts = get_clusters('4hhb.A')
    >>> print(clusts)
    {'pdbChain': {'@name': '2W72.A'}}

    """
    out = get_info(pdb_id, url_root =
'http://www.rcsb.org/pdb/rest/representatives?structureId=')
    out = to_dict(out)
    return remove_at_sign(out['representatives'])



def find_results_gen(search_term, field='title'):
    '''
    Return a generator of the results returned by a search of
    the protein data bank. This generator is used internally.

    Parameters
    ----------

    search_term : str
        The search keyword
```

```
    field : str
        The type of information to record about each entry

    Examples
    --------

    >>> result_gen = find_results_gen('bleb')
    >>> pprint.pprint([item for item in result_gen][:5])
    ['MYOSIN II DICTYOSTELIUM DISCOIDEUM MOTOR DOMAIN S456Y
BOUND WITH MGADP-BEFX',
     'MYOSIN II DICTYOSTELIUM DISCOIDEUM MOTOR DOMAIN S456Y
BOUND WITH MGADP-ALF4',
     'DICTYOSTELIUM DISCOIDEUM MYOSIN II MOTOR DOMAIN S456E
WITH BOUND MGADP-BEFX',
     'MYOSIN II DICTYOSTELIUM DISCOIDEUM MOTOR DOMAIN S456E
BOUND WITH MGADP-ALF4',
     'The structural basis of blebbistatin inhibition and specificity for myosin '
     'II']

    '''
    scan_params = make_query(search_term, querytype='AdvancedKeywordQuery')
    search_result_ids = do_search(scan_params)

    all_titles = []
    for pdb_result in search_result_ids:
        result= describe_pdb(pdb_result)
        if field in result.keys():
            yield result[field]

def parse_results_gen(search_term, field='title', max_results = 100, sleep_time=.1):
    '''
    Query the PDB with a search term and field while respecting the query frequency
     limitations of the API.

    Parameters
    ----------

    search_term : str
        The search keyword

    field : str
        The type of information to record about each entry

    max_results : int
        The maximum number of results to search through when
        determining the top results

    sleep_time : float
        Time (in seconds) to wait between requests. If this number is too small
        the API will stop working, but it appears to vary among different systems
```

```
    Returns
    -------

    all_data_raw : list of str

    '''

    if max_results*sleep_time > 30:
        warnings.warn("Because of API limitations, this function\
        will take at least " + str(max_results*sleep_time) + " seconds to return results.\
        If you need greater speed, try modifying the optional argument sleep_time=.1,
(although \
        this may cause the search to time out)" )

    all_data_raw = find_results_gen(search_term, field=field)
    all_data =list()
    while len(all_data) < max_results:
        all_data.append(all_data_raw.send(None))
        time.sleep(sleep_time)

    return all_data


def find_papers(search_term, **kwargs):
    '''
    Return an ordered list of the top papers returned by a keyword search of
    the RCSB PDB

    Parameters
    ----------

    search_term : str
        The search keyword

    max_results : int
        The maximum number of results to return

    Returns
    -------

    all_papers : list of strings
        A descending-order list containing the top papers associated with
        the search term in the PDB

    Examples
    --------

    >>> matching_papers = find_papers('crispr',max_results=3)
    >>> print(matching_papers)
```

```
    ['Crystal structure of a CRISPR-associated protein from thermus thermophilus',
    'CRYSTAL STRUCTURE OF HYPOTHETICAL PROTEIN SSO1404 FROM
SULFOLOBUS SOLFATARICUS P2',
    'NMR solution structure of a CRISPR repeat binding protein']

    '''
    all_papers = parse_results_gen(search_term, field='title', **kwargs)
    return remove_dupes(all_papers)

def find_authors(search_term, **kwargs):
    '''Return an ordered list of the top authors returned by a keyword search of
    the RCSB PDB

    This function is based on the number of unique PDB entries a given author has
    his or her name associated with, and not author order or the ranking of the
    entry in the keyword search results. So if an author tends to publish on topics
    related to the search_term a lot, even if those papers are not the best match for
    the exact search, he or she will have priority in this function over an author
    who wrote the one paper that is most relevant to the search term. For the latter
    option, just do a standard keyword search using do_search.

    Parameters
    ----------

    search_term : str
        The search keyword

    max_results : int
        The maximum number of results to return

    Returns
    -------

    out : list of str


    Examples
    --------

    >>> top_authors = find_authors('crispr',max_results=100)
    >>> print(top_authors[:10])
    ['Doudna, J.A.', 'Jinek, M.', 'Ke, A.', 'Li, H.', 'Nam, K.H.']

    '''

    all_individuals = parse_results_gen(search_term, field='citation_authors',
**kwargs)

    full_author_list = []
    for individual in all_individuals:
```

```
        individual = individual.replace('.,', '.;')
        author_list_clean = [x.strip() for x in individual.split(';')]
        full_author_list+=author_list_clean

    out = list(chain.from_iterable(repeat(ii, c) for ii,c in
Counter(full_author_list).most_common()))

    return remove_dupes(out)

def find_dates(search_term, **kwargs):
    '''
    Return an ordered list of the PDB submission dates returned by a
    keyword search of the RCSB PDB. This can be used to assess the
    popularity of a gievne keyword or topic

    Parameters
    ----------

    search_term : str
        The search keyword

    max_results : int
        The maximum number of results to return

    Returns
    -------

    all_dates : list of str
        A list of calendar strings associated with the search term, these can
        be converted directly into time or datetime objects

    '''
    all_dates = parse_results_gen(search_term, field='deposition_date', **kwargs)
    return all_dates



def list_taxa(pdb_list, sleep_time=.1):
    '''Given a list of PDB IDs, look up their associated species

    This function digs through the search results returned
    by the get_all_info() function and returns any information on
    taxonomy included within the description.

    The PDB website description of each entry includes the name
    of the species (and sometimes details of organ or body part)
    for each protein structure sample.

    Parameters
    ----------
```

```
    pdb_list : list of str
        List of PDB IDs

    sleep_time : float
        Time (in seconds) to wait between requests. If this number is too small
        the API will stop working, but it appears to vary among different systems

    Returns
    -------

    taxa : list of str
        A list of the names or classifictions of species
        associated with entries

    Examples
    --------

    >>> crispr_query = make_query('crispr')
    >>> crispr_results = do_search(crispr_query)
    >>> print(list_taxa(crispr_results[:10]))
    ['Thermus thermophilus',
     'Sulfolobus solfataricus P2',
     'Hyperthermus butylicus DSM 5456',
     'unidentified phage',
     'Sulfolobus solfataricus P2',
     'Pseudomonas aeruginosa UCBPP-PA14',
     'Pseudomonas aeruginosa UCBPP-PA14',
     'Pseudomonas aeruginosa UCBPP-PA14',
     'Sulfolobus solfataricus',
     'Thermus thermophilus HB8']


    '''

    if len(pdb_list)*sleep_time > 30:
        warnings.warn("Because of API limitations, this function\
        will take at least " + str(len(pdb_list)*sleep_time) + " seconds to return results.\
        If you need greater speed, try modifying the optional argument sleep_time=.1,
(although \
        this may cause the search to time out)" )

    taxa = []

    for pdb_id in pdb_list:
        all_info = get_all_info(pdb_id)
        species_results = walk_nested_dict(all_info, 'Taxonomy',
maxdepth=25,outputs=[])
        first_result = walk_nested_dict(species_results,'@name',outputs=[])
        if first_result:
            taxa.append(first_result[-1])
```

```
            else:
                taxa.append('Unknown')

            time.sleep(sleep_time)

        return taxa

def list_types(pdb_list, sleep_time=.1):
    '''Given a list of PDB IDs, look up their associated structure type


    Parameters
    ----------

    pdb_list : list of str
        List of PDB IDs

    sleep_time : float
        Time (in seconds) to wait between requests. If this number is too small
        the API will stop working, but it appears to vary among different systems


    Returns
    -------

    infotypes : list of str
        A list of the structure types associated with each PDB
        in the list. For many entries in the RCSB PDB, this defaults
        to 'protein'

    Examples
    --------

    >>> crispr_query = make_query('crispr')
    >>> crispr_results = do_search(crispr_query)
    >>> print(list_types(crispr_results[:5]))
    ['protein', 'protein', 'protein', 'protein', 'protein']
    '''

    if len(pdb_list)*sleep_time > 30:
        warnings.warn("Because of API limitations, this function\
        will take at least " + str(len(pdb_list)*sleep_time) + " seconds to return results.\
        If you need greater speed, try modifying the optional argument sleep_time=.1,
(although \
        this may cause the search to time out)" )

    infotypes = []
    for pdb_id in pdb_list:
        all_info = get_all_info(pdb_id)
        type_results = walk_nested_dict(all_info, '@type', maxdepth=25,outputs=[])
```

```
            if type_results:
                infotypes.append(type_results[-1])
            else:
                infotypes.append('Unknown')
            time.sleep(sleep_time)

        return infotypes


'''
==================
Helper Functions
==================
'''

def to_dict(odict):
    '''Convert OrderedDict to dict

    Takes a nested, OrderedDict() object and outputs a
    normal dictionary of the lowest-level key:val pairs

    Parameters
    ----------

    odict : OrderedDict

    Returns
    -------

    out : dict

        A dictionary corresponding to the flattened form of
        the input OrderedDict

    '''

    out = loads(dumps(odict))
    return out

def remove_at_sign(kk):
    '''Remove the '@' character from the beginning of key names in a dict()

    Parameters
    ----------

    kk : dict
        A dictionary containing keys with the @ character
        (this pops up a lot in converted XML)

    Returns
```

```
    -------

    kk : dict (modified in place)
        A dictionary where the @ character has been removed

    '''
    tagged_keys = [thing for thing in kk.keys() if thing.startswith('@')]
    for tag_key in tagged_keys:
        kk[tag_key[1:]] = kk.pop(tag_key)

    return kk

def remove_dupes(list_with_dupes):
    '''Remove duplicate entries from a list while preserving order

    This function uses Python's standard equivalence testing methods in
    order to determine if two elements of a list are identical. So if in the list [a,b,c]
    the condition a == b is True, then regardless of whether a and b are strings, ints,
    or other, then b will be removed from the list: [a, c]

    Parameters
    ----------

    list_with_dupes : list
        A list containing duplicate elements

    Returns
    -------
    out : list
        The list with the duplicate entries removed by the order preserved


    Examples
    --------
    >>> a = [1,3,2,4,2]
    >>> print(remove_dupes(a))
    [1,3,2,4]

    '''
    visited = set()
    visited_add = visited.add
    out = [ entry for entry in list_with_dupes if not (entry in visited or
visited_add(entry))]
    return out

def walk_nested_dict(my_result, term, outputs=[], depth=0, maxdepth=25):
    '''
    For a nested dictionary that may itself comprise lists of
    dictionaries of unknown length, determine if a key is anywhere
    in any of the dictionaries using a depth-first search
```

```
    Parameters
    ----------

    my_result : dict
       A nested dict containing lists, dicts, and other objects as vals

    term : str
       The name of the key stored somewhere in the tree

    maxdepth : int
       The maximum depth to search the results tree

    depth : int
       The depth of the search so far.
       Users don't usually access this.

    outputs : list
       All of the positive search results collected so far.
       Users don't usually access this.

    Returns
    -------

    outputs : list
       All of the search results.

    '''

    if depth > maxdepth:
       warnings.warn('Maximum recursion depth exceeded. Returned None for the
search results,'+
                     ' try increasing the maxdepth keyword argument.')
       return None


    depth = depth + 1

    if type(my_result)==dict:
       if term in my_result.keys():
          outputs.append(my_result[term])

       else:
          new_results = list(my_result.values())
          walk_nested_dict(new_results, term, outputs=outputs,
depth=depth,maxdepth=maxdepth)

    elif type(my_result)==list:
       for item in my_result:
          walk_nested_dict(item, term, outputs=outputs,
depth=depth,maxdepth=maxdepth)
```

```
    else:
        pass
        # dead leaf

    # this conditional may not be necessary
    if outputs:
        return outputs
    else:
        return None
```

Bibliography

1. S. Bryson, D. Kenwright, M. Cox, D. Ellsworth, R. Haimes, Visually exploring gigabyte data sets in real time. Commun. ACM **42**(8), 82–90 (1999)
2. P. Lyman, H.R. Varian, *How Much Information*. University of California at Berkeley, 2017 (2000) (Online). Available: http://www2.sims.berkeley.edu/research/projects/how-much-info/
3. S. Sicular, *Gartner's Big Data Definition Consists of Three Parts, Not to Be Confused with Three "V"s*, *Forbes, 2013*. Available: http://www.forbes.com/sites/gartnergroup/2013/03/27/gartners-big-data-definition-consists-of-three-parts-not-to-be-confused-with-three-vs/
4. Y. Demchenko, P. Grosso, C. de Laat, P. Membrey, Addressing big data issues in scientific data infrastructure, in *2013 International Conference on Collaboration Technologies and Systems (CTS)*, San Diego, CA, 2013, pp. 48–55
5. A. Katal, M. Wazid, R.H. Goudar, Big data: issues, challenges, tools and good practices, in *2013 Sixth International Conference on Contemporary Computing* (IC3) 2013, pp. 404–409
6. R. Agrawal, Big data and its application, in *2014 Conference on IT in Business, Industry and Government (CSIBIG)*, Indore, 2014, p. 1
7. C. Anderson, The end of theory: the data deluge makes the scientific method obsolete. Wired Mag., 1607 (2008)
8. J. Gueyoung, N. Gnanasambandam, T. Mukherjee, Synchronous parallel processing of big-data analytics services to optimize performance in federated clouds, in *IEEE 5th International Conference on Cloud Computing (CLOUD)*, 2012, pp. 811–818
9. A. Rajpurohit, Big data for business managers—bridging the gap between potential and value, in *2013 IEEE International Conference on Big Data*, Silicon Valley, CA, 2013, pp. 29–31
10. A. Vera-Baquero, R. Colomo-Palacios, O. Molloy, Business process analytics using a big data approach. IT Prof. **15**(6), 29–35 (2013)
11. M. Bada, L. Hunter, Enrichment of OBO ontologies. J. Biomed. Inform. **40**(3), 300–315 (2007)
12. E. Camon, M. Magrane, D. Barrell, V. Lee, E. Dimmer, J. Maslen et al., The gene ontology annotation (GOA) database: sharing knowledge in Uniprot with gene ontology. Nucleic Acids Res **32**, D262–D266 (2004)
13. P. Buneman, S.B. Davidson, K. Hart, G.C. Overton, L. Wong, A data transformation system for biological data sources, in *Proceedings of the 21th International Conference on Very Large Data Bases*, 1995, pp. 158–169
14. E. Pennisi, Genome data shake tree of life. Science **280**(5364), 672–674 (1998)

C. Y. Cho et al., *Large Scale Data Analytics*,
Studies in Computational Intelligence 806,
https://doi.org/10.1007/978-3-030-03892-2

15. A.S. Sidhu, M. Bellgard, Protein data integration problem, in *Biomedical Data and Applications*, ed. by A.S. Sidhu, T.S. Dillon (Springer, Berlin, 2009), pp. 55–69
16. Z. Lacroix, Issues to address while designing a biological information system, in *Bioinformatics: Managing Scientific Data*, ed. by Z. Lacroix, T. Critchlow. The Morgan Kaufmann Series in Multimedia Information and Systems, 2003, pp. 75–108
17. A. Kadadi, R. Agrawal, C. Nyamful, R. Atiq, Challenges of data integration and interoperability in big data, in *2014 IEEE International Conference on Big Data (Big Data)*, Washington, DC, 2014, pp. 38–40
18. K. Baclawski, M. Kokar, P. Kogut, L. Hart, J. Smith, W. Holmes, III et al., Extending UML to support ontology engineering for the semantic web, in *≪UML≫ 2001—The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, vol. 2185, ed. by M. Gogolla, C. Kobryn (Springer, Berlin, 2001), pp. 342–360
19. A. Doan, A.Y. Halevy, Semantic integration research in the database community: a brief survey. AI Mag. **26**, 83 (2005)
20. T.R. Gruber, A translation approach to portable ontology specifications. Knowl. Acquis. **5** (2), 199–220 (1993)
21. A.S. Sidhu, M. Bellgard, T.S. Dillon, Classification of information about proteins, in *Bioinformatics: Tools and Applications*, ed. by D. Edwards, J. Stajich, D. Hansen (Springer, New York, 2009), pp. 243–258
22. M. Uschold, M. King, Towards a methodology for building ontologies. Workshop on Basic Ontological Issues in Knowledge Sharing Held in Conjunction with IJCAI 1995 (Morgan Kaufmann, 1995)
23. M. Uschold, M. Gruninger, Ontologies: principles methods and applications. Knowl. Eng. Rev. **11**(2), 93–155 (1996)
24. M. Uschold, M. King, S. Morale, Y. Zorgios, The enterprise ontology. Knowl. Eng. Rev. **13** (1), 31–89 (1998)
25. M. Gruninger, M.S. Fox, Methodology for design and evaluation of ontologies. Workshop on Basic Ontological Issues in Knowledge Sharing Held in Conjunction with IJCAI 1995, Montreal, Canada (Morgan Kaufmann, 1995)
26. S. Staab, R. Studer, H.P. Schnurr, Y. Sure, Knowledge processes and ontologies. IEEE Intell. Syst. **16**(1), 26–34 (2001)
27. M. Genesereth, Knowledge interchange format, in *Second International Conference on Principles of Knowledge Representation and Reasoning*, Cambridge (Morgan Kaufmann, 1991)
28. M. Genesereth, R. Fikes, *Knowledge Interchange Format Version 3 Reference Manual* (Stanford University Logic Group, Stanford, 1992)
29. Ontoweb, A survey on methodologies for developing, maintaining, evaluating and reengineering ontologies, in *Deliverable 1.4 of OntoWeb Project*, ed. by M. Fernández-López, Karlsruhe, Germany, AIFB Germany & VUB STAR Lab (2002). Available: http://www.ontoweb.org/About/Deliverables/index.html
30. G. Schreiber, H. Akkermans, A. Anjewierden, R. Dehoog, N. Shadbolt, W. Vandevelde, B. Wielinga, Knowledge Engineering and Management: The Common KADS Methodology (MIT Press, Cambridge, 2000)
31. D.L. McGuinness, F. Harmelen (eds.), W3C-OWL, OWL web ontology language overview, in *W3C Recommendation 10 February 2004, McGuinness*. World Wide Web Consortium (2004)
32. N. Arch-int, S. Arch-int, Semantic information integration for electronic patient records using ontology and web services model, in *2011 International Conference on Information Science and Applications*, Jeju Island, 2011, pp. 1–7
33. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, K. Wilkinson, Jena: implementing the semantic web recommendations, in *Proceedings of the 13th World Wide Web Conference*, New York City, USA, pp. 74–83, 17–22 May 2004

34. X. Liu, C. Hu, J. Huang, F. Liu, OPSDS: a semantic data integration and service system based on domain ontology, in *2016 IEEE First International Conference on Data Science in Cyberspace (DSC)*, Changsha, 2016, pp. 302–306
35. W. Yunxiao, Z. Xuecheng, The research of multi-source heterogeneous data integration based on LINQ, in *2012 International Conference on Computer Science and Electronics Engineering (ICCSEE)*, 2012, pp. 147–150
36. Querying Across Relationships (LINQ to SQL), Microsoft, 2017 (Online). Available: https://msdn.microsoft.com/en-us/library/vstudio/bb386932(v=vs.100).aspx
37. What is Python? Executive Summary, Python.org, 2017 (Online). Available: https://www.python.org/doc/essays/blurb/
38. E.J. Qaisar, Introduction to cloud computing for developers: key concepts, the players and their offerings, in *2012 IEEE TCF Information Technology Professional Conference*, Ewing, NJ, 2012, pp. 1–6
39. M. Hamdaqa, L. Tahvildari, Cloud computing uncovered: a research landscape. Adv. Comput., 41–85 (2012)
40. A. Hejlsberg, M. Torgersen, The .NET standard query operators, Microsoft, 2017 (Online). Available: http://msdn.microsoft.com/en-us/library/bb394939.aspx
41. W. Gilpin, A python API for the RCSB protein data bank (PDB), Github, 2016 (Online). Available: https://github.com/williamgilpin/pypdb