The SHARE System
A Semantic Web Based Approach for Evaluating Queries Across Distributed Bioinformatics Databases and Software
by
Ben Vandervalk

B. Math. (Computer Science), The University of Waterloo, 2004

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in
The Faculty of Graduate Studies
(Bioinformatics)

THE UNIVERSITY OF BRITISH COLUMBIA
(Vancouver)
April, 2011
© Ben Vandervalk 2011
Abstract

Many bioinformatics studies require combined use of data sets and software developed by different research labs. At the current time, accomplishing such studies requires the development of custom scripts that act as “glue” for the independent resources, performing transformations on the data sets that will allow them to be loaded into a single database and/or shuttled through different pieces of software. Due to the tedium and inefficiency of manual data/software integration, many institutions and research groups have sought to find a more reliable and automatic approach. The most significant integration project in recent years has been the Semantic Web activity of the World Wide Web Consortium (W3C), which aims to automate data integration not only in bioinformatics, but on the WWW as a whole. The goal of the Semantic Web is to interlink data on the web in a manner that is similar to the way that HTML pages are linked, while at the same time making the data available in a universal form that can be easily processed by software. In this thesis, the author describes a distributed query system called SHARE (Semantic Health and Research Environment) which demonstrates how the available standards and tools of the Semantic Web can be assembled into a framework for automating data and software integration in bioinformatics. We find that while SHARE has a similar architecture to existing query systems, the use of Semantic Web technologies has important advantages for the implementation, maintenance, and addition of new data sources to the system. After reviewing the mechanics of SHARE, we examine the crucial problem of optimizing queries in an environment where statistics about the data sources are typically not available. A query evaluation procedure called GREEDY is presented that addresses this challenge by: i) interleaving the planning and execution phases of a query, and ii) learning statistics from the execution of previous queries. We conclude by highlighting the unique strengths of SHARE and GREEDY in relation to other integration systems, and review important areas for future work.
Preface

The design of SHARE was a joint effort of the Wilkinson laboratory. The majority of the programming work for the system, including the core query engine and the live web demonstration, was done by Luke McCarthy, a full-time employee of the Wilkinson laboratory. My own contributions to the programming were the query optimizer and the ability to query SPARQL endpoints as data sources.


Chapter 3, which describes the optimization algorithm for SHARE, is a further development of work published in: Ben P. Vandervalk, E. Luke McCarthy, and Mark D. Wilkinson, “Optimization of Distributed SPARQL Queries Using Edmonds’ Algorithm and Prim’s Algorithm,” in 12th IEEE International Conference on Computational Science and Engineering (CSE 2009), vol. 4, pp. 29-31, 2009. In particular, the idea for the GREEDY algorithm is based on Section 7 of that paper, titled “Adaptive query execution using Prim’s algorithm”. The text and figures for Chapter 3 were created from scratch.
# Table of Contents

Abstract ................................................................. [i]
Preface ................................................................. [iii]
Table of Contents ....................................................... [iv]
List of Tables .......................................................... [vi]
List of Figures .......................................................... [vii]
Acknowledgements ....................................................... [viii]

1 Introduction .......................................................... 1
   1.1 The Motivation for Data and Software Integration in Bioinformatics 1
   1.2 Fundamental Problems of Data and Software Integration .......... 2
   1.3 Integration Technologies and Related Research Projects ......... 4
      1.3.1 Data Warehouses ........................................ 4
      1.3.2 XML .................................................. 4
      1.3.3 The Semantic Web .................................... 7
      1.3.4 Web Services ......................................... 14
      1.3.5 Mediator Systems ................................... 17
   1.4 Thesis Outline ................................................. 19

2 SHARE System Architecture ....................................... 20
   2.1 Introduction .................................................. 20
   2.2 The SPARQL Query Language ................................ 20
   2.3 SHARE Data Sources ......................................... 23
      2.3.1 SADI Services ....................................... 23
      2.3.2 SPARQL Endpoints .................................. 24
   2.4 SHARE Query Resolution .................................... 29
      2.4.1 Overview ............................................ 29
      2.4.2 Resolution Process of an Example SHARE Query .......... 29
      2.4.3 Pseudocode for SHARE Query Resolution ............... 31
      2.4.4 Validating Inputs for Services ........................ 32
   2.5 SHARE Online Demonstration ................................ 35
   2.6 Source Code .................................................. 35
   2.7 Discussion .................................................... 35

3 An Adaptive Query Evaluation Algorithm for SHARE .......... 39
   3.1 Introduction .................................................. 39
   3.2 Related Work ................................................ 40
   3.3 Key Points of SHARE Query Resolution ................. 41
      3.3.1 Resolving Triple Patterns ........................... 41
      3.3.2 OWL Inferences About Predicates ..................... 41


Table of Contents

3.4 The Distributed SPARQL Query Optimization Problem .................................. 42
   3.4.1 An Illustrative Example ........................................................................... 42
   3.4.2 Assumptions ............................................................................................. 45
3.5 Secondary Optimization: Intersections of Variable Bindings .......................... 46
3.6 The GREEDY Optimization Algorithm ......................................................... 51
   3.6.1 Overview ................................................................................................. 51
   3.6.2 Cost Estimation for Query Patterns ......................................................... 51
3.7 Evaluation ...................................................................................................... 54
   3.7.1 Evaluation Procedure .............................................................................. 54
   3.7.2 Results ..................................................................................................... 60
3.8 Future Work .................................................................................................... 73
3.9 Conclusion ....................................................................................................... 73
3.10 Source Code .................................................................................................. 74

4 Conclusion ......................................................................................................... 75

Bibliography .......................................................................................................... 77

Appendices

A Supporting Material for Chapter 2 ................................................................. 90
   A.1 An Example RDF Service Description for a SADI Service ..................... 90

B Supporting Material for Chapter 3 ................................................................. 91
   B.1 Variants of Training Queries ..................................................................... 91
      B.1.1 Variants of Query 1 ............................................................................... 91
      B.1.2 Variants of Query 3 ............................................................................... 92
      B.1.3 Variants of Query 5 ............................................................................... 93
   B.2 Randomly Generated Orderings for Test Queries ................................. 94
      B.2.1 Orderings for Query 2 .......................................................................... 94
      B.2.2 Orderings for Query 4 .......................................................................... 96
      B.2.3 Orderings for Query 6 .......................................................................... 98
# List of Tables

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Results for example SPARQL SELECT query</td>
<td>20</td>
</tr>
<tr>
<td>2.2</td>
<td>List of SPARQL endpoints currently indexed by SHARE</td>
<td>27</td>
</tr>
<tr>
<td>2.3</td>
<td>Pointers to SHARE source code</td>
<td>36</td>
</tr>
<tr>
<td>3.1</td>
<td>Description of execution steps for an inefficient query ordering</td>
<td>42</td>
</tr>
<tr>
<td>3.2</td>
<td>Description of execution steps for an efficient query ordering</td>
<td>45</td>
</tr>
<tr>
<td>3.3</td>
<td>Description of execution steps without the variable bindings optimization</td>
<td>50</td>
</tr>
<tr>
<td>3.4</td>
<td>Description of execution steps with the variable bindings optimization</td>
<td>50</td>
</tr>
<tr>
<td>3.5</td>
<td>Description of execution steps for Query 2 under BASIC</td>
<td>68</td>
</tr>
<tr>
<td>3.6</td>
<td>Description of execution steps for Query 2 under untrained GREEDY</td>
<td>68</td>
</tr>
<tr>
<td>3.7</td>
<td>Description of execution steps for Query 2 under trained GREEDY</td>
<td>69</td>
</tr>
<tr>
<td>3.8</td>
<td>Description of execution steps for Query 6 under BASIC</td>
<td>72</td>
</tr>
<tr>
<td>3.9</td>
<td>Description of execution steps for Query 6 under untrained GREEDY</td>
<td>72</td>
</tr>
<tr>
<td>3.10</td>
<td>Description of execution steps for Query 6 under trained GREEDY</td>
<td>72</td>
</tr>
</tbody>
</table>
# List of Figures

1.1 Example illustrating the problem of merging two XML datasets ........................................... 6
1.2 An example RDF dataset ............................................................................................................ 8
1.3 Example of a successful RDF-merge .......................................................................................... 10
1.4 Example of an unsuccessful RDF-merge .................................................................................... 11
1.5 Using different OWL ontologies with the same RDF dataset .................................................. 14
1.6 Example of a Taverna workflow .................................................................................................. 16
1.7 Mediator architecture .................................................................................................................. 18

2.1 An example RDF dataset, to illustrate SPARQL usage ............................................................ 21
2.2 An example SPARQL SELECT query ............................................................................................ 21
2.3 An example SPARQL CONSTRUCT query .................................................................................. 22
2.4 An example SADI service .......................................................................................................... 23
2.5 Map of generated properties for SADI services ........................................................................ 24
2.6 Querying of remote SPARQL endpoints in SHARE .................................................................. 26
2.7 A map of generated properties for SPARQL endpoints ............................................................... 28
2.8 An overview of SHARE query resolution .................................................................................... 30
2.9 Screenshot of the SHARE demonstration site ........................................................................... 36

3.1 Example of an inefficient ordering of triple patterns ................................................................. 43
3.2 Example of an efficient ordering of triple patterns ..................................................................... 44
3.3 Pseudocode for the variable bindings optimization ................................................................. 47
3.4 Query to demonstrate the variable bindings optimization ......................................................... 48
3.5 Execution plans with and without the variable bindings optimization ...................................... 49
3.6 Example regression line to determine baseTime and timePerInput ......................................... 53
3.7 Six example SPARQL queries for evaluating GREEDY ............................................................ 54
3.9 State of predicate statistics during experiment .......................................................................... 64
3.10 Execution plans for Query 2 under BASIC, untrained GREEDY, and trained GREEDY ............ 67
3.11 Execution plans for Query 6 under BASIC, untrained GREEDY, and trained GREEDY .......... 71
Acknowledgements

Thanks to my supervisor, Mark Wilkinson, for his unfaltering support, patience, and enthusiasm, and thanks to Luke McCarthy for helping me to build off of his excellent work. Thanks to Paul Pavlidis and Laks Lakshmanan, who have sacrificed their time to be on my advisory committee and to review this thesis.

This research was funded by the Heart and Stroke Foundation of BC and Yukon, and additional support for the development of SHARE was also provided by CIHR and NSERC. Ongoing development and promotion of the SADI web service standard is made possible by CANARIE. All research of the Wilkinson Lab is supported by the James Hogg Research Centre of the Providence Heart and Lung Institute.
Chapter 1

Introduction

1.1 The Motivation for Data and Software Integration in Bioinformatics

The development of high-throughput methods in molecular biology, such as next generation sequencing [1], microarrays [2], and chromatin immunoprecipitation [3], has led to an explosion in the quantity and diversity of biological data that is available on the web. In turn, the new data has driven the development of new computational tools for tasks such as gene prediction [4] and protein structure prediction [5, 6]. The rapid growth of bioinformatics data and software is evidenced by the annual “database issue” and “web server issue” of the Nucleic Acids Research (NAR) journal, which have now published papers describing ∼ 1200 online databases and ∼ 1400 online analysis tools, respectively. NAR provides an online list of the databases [7], while the analysis tools have been catalogued by the Bioinformatics Links Directory [8].

One of the most intriguing things about the body of available bioinformatics resources is the large number of logical interconnections. In general, the resources describe entities that are closely related (such as genes [9], transcripts [10], and proteins [11]) or describe different aspects of the same entities (such as the sequence [11], localization [12–13], interaction partners [14–16], or structure of a protein [17]). As such, there are many motivations for integrating datasets and software that have been produced by different laboratories. Some applications of integration include:

assembling comprehensive datasets For most studies or reviews, researchers will want to obtain the most complete dataset possible, which usually requires extracting and merging data from many publications and websites. This need has led to the development of many databases that play the role of shared repositories for a single type of information. Such repositories exist for mutations in cancer [18], antibody sequences [19], protein interactions [15–16, 20], protein structures [17], drug targets [21], microarray experiments [22–24], and biological pathways [25–27], to name a few.

combining evidence Many activities in bioinformatics involve combining evidence in order to increase the confidence of predictions. One such task is the annotation of a newly-sequenced genome, where researchers must integrate direct experimental evidence, computational predictions, and known information about related organisms in order to identify gene-coding regions, non-coding RNAs, synteny blocks, and putative protein functions [28]. There are many other areas of study for which the integration of evidence is essential, such as the reconstruction of phylogenetic trees [29], the prediction of subcellular protein localization [30], and the prediction of protein-protein interactions [20, 31].

interpreting experimental results Genome-wide association studies [32] and microarray experiments identify potentially long lists of genes that have a significant role in a particular disease or experimental condition. However, these experiments do not explain why these “hits” are significant or how they are biologically related. The researcher may have a
large number of questions about the top-ranked genes which require external datasets or analysis, such as:

⇒ “For what other diseases or conditions was this gene found to be significant?”
⇒ “What papers have been published on this gene?”
⇒ “What metabolic, signaling, and regulatory pathways does this gene participate in?”
⇒ “Where do the protein product(s) of this gene localize within the cell?”
⇒ “What other tissues is the gene transcribed in?”
⇒ “What are known interaction partners for the protein product(s) of this gene?”
⇒ “Is this gene alternatively spliced?”
⇒ . . .

Such contextual information is crucial for forming hypotheses and for prioritizing genes for further study or validation. In fact, several data integration systems have been designed specifically for prioritizing gene lists [33–35].

**comparing experiments** Meta-analyses of microarray experiments are an important example of data integration in bioinformatics. The most common goal of such analyses is to validate differentially expressed genes for a certain disease or set of experimental conditions across experiments [36]; this method of validation is promising because it is significantly more cost-effective than experimentally validating results by PCR. Other types of meta-analyses have been conducted to determine the reliability of microarray results across platforms [37] and to validate pairs of coexpressed genes [38].

**executing multi-step data analyses** Many analyses in bioinformatics are conceptualized as pipelines or workflows, in which each step performs a different transformation on the data. For instance, a “first-pass” phylogenetic tree for a gene or protein is typically constructed by: (i) identifying the homologs of the gene/protein with a BLAST search, (ii) creating a multiple sequence alignment (MSA) of the homologs, and (iii) using a phylogenetic tree program to infer evolutionary distances from the MSA [39]. Other tasks that are frequently implemented as pipelines are genome annotation, drug discovery, and microarray analysis.

### 1.2 Fundamental Problems of Data and Software Integration

The current body of websites, databases, and software that is available to bioinformaticians has been produced by a large number of research groups acting independently and without a shared set of standards. This has inevitably resulted in a large number of incompatible schemas and software interfaces that make the combined use of resources unnecessarily difficult and time-consuming [40]. As the field has grown, the bioinformatics community has recognized the need for standardization, and many XML formats have been developed for the exchange of data within specific domains such as microarrays, molecular interactions, and biological pathways [41]. Unfortunately, there are several integration-related problems which XML alone cannot solve, as will be discussed in Section 1.3.

While all of the problems related to data and software integration can be (and typically are) solved manually on a case-by-case basis, the principal goal of integration systems is to resolve these problems automatically. In particular, the main tasks that integration systems seek to automate are:
1.2. Fundamental Problems of Data and Software Integration

discovery of relevant resources The first of step of integration is to identify the available
databases and software that need to be combined for the desired analysis. In current prac-
tice, resources are usually discovered manually through internet search engines, literature,
and link directories.

matching entities across datasets Whenever two related datasets are combined, the com-
mon entities (e.g. genes) between those datasets must be identified and merged. The ideal
solution to this problem is to establish a system of globally-shared identifiers for biological
entities; however, no such system yet exists. The LSID [42] identifier and resolution scheme
was recently proposed for this purpose, but unfortunately it has failed to gain widespread
acceptance by the community. LSID identifiers were based on a URN scheme, and aimed
to support guaranteed persistence of identifiers and immutability of data records, among
other features. The most common objection to the LSID system was that the resolution of
identifiers required specialized software and could not be performed with a standard web
browser (for example, see [42, 43]). In the absence of a global identifiers, more sophisticated
methods must be employed for matching entities across datasets, based on the content of
the records. A variety of methods for entity matching have been studied within the field
of database research, including rule-based systems, decision trees, and other supervised
learning methods [44].

schema integration Data models such as the relational model, XML, and RDF impose only
general constraints on the structure of a dataset. Under these models, a dataset must be
encoded as a set of cross-referenced tables, a tree, or a directed graph, respectively. Any
remaining decisions about structure are left to the data owner, and so the same dataset
may be organized according to a multitude of different schemas. A fundamental problem
arises when two parties wish to share datasets that are logically related, but whose schemas
are mutually incompatible. Data cannot be transferred directly from one database to the
other, but must first be mapped between the two schemas. Typically this mapping must
be worked out by hand, and the complexity of the problem grows with the number of
databases to be integrated. Schema integration is the central problem of data integration,
and has been widely studied in the context of relational databases [45], XML (e.g. [46]),
and OWL [47].

connecting software interfaces In order to accomplish multi-step analyses, an integration
system must be able to route data from the output of one program to the input of another
in an automated fashion. Pipes[48] are a well-known facility of the Unix command line
that was designed precisely for this purpose. A standard Unix program reads its input data
from a special file descriptor called STDIN, and writes its output to a special file descriptor
called STDOUT. When a series of programs is connected by Unix pipes, the STDOUT of
each program is sent to the STDIN of the next, thereby creating a continuous processing
pipeline. While the Unix command line is a sturdy and versatile tool, specialized workflow
systems have been developed for bioinformatics that go beyond the functionality of Unix
pipes in a number of ways. Typically, the functional units of such systems are Web Ser-
vices rather than locally installed programs, and this obviates the need for downloading,
installing, and configuring software on the user’s local machine. More importantly, work-
flow frameworks automate (or partially automate) a number of fundamental tasks such
as service discovery, service invocation, identification of services that may be connected,
and execution of an overall pipeline. Beyond this, the most advanced workflow systems
are capable of translating an abstract, query-based representation of a workflow into a
concrete implementation. SHARE, the subject of this thesis, is one such system.
1.3 Integration Technologies and Related Research Projects

In this section, we introduce the main technologies that have been developed for data and software integration, discuss the advantages and disadvantages of each, and highlight related systems that have been developed for use in bioinformatics.

1.3.1 Data Warehouses

Conceptually, warehousing is the most straightforward approach to data integration. Data warehouses are constructed by combining a set of related sources into a single database with a unified schema. Beyond the task of designing an unified schema, the creators of a warehouse must develop scripts for extracting data from the original sources, transforming the data to fit the schema, and loading the data into the warehouse. In the database literature, this is known as the Extract-Transform-Load (ETL) cycle. Procedures for extraction and transformation must typically be written by hand after detailed study of the original sources.

Warehousing is a popular methodology in bioinformatics, and numerous warehouses have been constructed for specific types of data such as structural families of proteins, sequences of fungal species, ligands, and crop-related data, to name a few. In addition, several warehouses have been built to integrate popular databases such as GenBank, UniProt, and GO; two warehouses of this type are Atlas and SeqHound. The most widely known warehousing system in biology is the Sequence Retrieval System (SRS). SRS is a framework for integrating databases that are encoded as collections of plain text files (one file per record), and can be configured to parse a wide variety of syntaxes. The principal functionality offered by SRS is the ability to execute efficient, field-specific keyword searches across a large collection of databases. (EBI maintains a publicly-accessible installation of SRS that currently includes 280 different databases.) Some recent data warehousing efforts have utilized RDF as their data model, such as the Neurocommons Knowledge Base and the Pathway Knowledge Base; this approach serves to simplify some aspects of the schema integration problem, as will be discussed in Section 1.3.3.

The warehousing approach to integration is often compared to the opposing methodology of view integration that is used by mediator systems; mediator systems function by translating a user query into a number of subqueries that are issued against the databases in their native locations. The principal advantages of warehouses over mediators are speed and reliability; warehouse users need not rely on the availability or performance of numerous remote data sources, and query processing is not delayed by the transfer of data across the network. A further advantage of warehouses is that the provider has complete control over the organization of the data. This allows him/her/them to include additional processing steps that result in a cleaner and simpler database, such as filtering, entity-matching, and translation to a uniform identifier scheme. However, centralized control is also a disadvantage in the sense that it places the full burden of maintenance on the warehouse provider. Warehouses must be updated regularly from the original sources, and data extraction scripts must be repaired whenever the schemas of the original sources change.

1.3.2 XML

XML (eXtensible Markup Language) is a framework for defining application-specific data formats. All XML files are text-based and share a general syntax consisting of tags (URIs enclosed by angle brackets) and content organized in a hierarchical manner. Listing 1.1 shows an example XML file that encodes pathway information, in accordance to a hypothetical schema called PathwayML. (Hypothetical schemas are used in this section to avoid unnecessarily complicated examples; however, the examples used here are valid XML.) The central standard of
1.3. Integration Technologies and Related Research Projects

XML is XML Schema\[46\], which allows the user to rigorously specify the rules of an XML format. An XML schema primarily specifies structural relationships between tags, such as the parent/child relationships, the relative order of sibling tags, and the number of legal occurrences for a tag in a given context. It also describes the type of content (i.e. datatype) for each tag, and specifies an analogous set of rules for XML attributes (such as “reactionID” in listing 1.1). Listing 1.2 shows the XML Schema for the PathwayML file of listing 1.1.

Listing 1.1: Example XML file describing part of a metabolic pathway

```xml
<?xml version="1.0" encoding="UTF-8"?>
<pathway pathwayID="KEGG:hsa00234" xmlns="http://www.pathwayml.org/schema">
  <reaction reactionID="KEGG:R00335">
    <enzyme enzymeID="UniProt:P04035">
      <substrate>PubChem:3649</substrate>
      <product>PubChem:3708</product>
    </enzyme>
    <reaction>
    </reaction>
  </reaction>
</pathway>
```

Listing 1.2: Example XML Schema (“PathwayML”) describing the XML file of listing 1.1

One of the principal features of XML Schema is that schema files are “machine readable”. This means that, in contrast to a natural language description of the schema, an XML Schema file has a simple and rigid structure that makes automated processing by software a relatively straightforward task. (XML data files are themselves machine readable, as are many other formats such as ASN.1 [62] and RDF [63].) The machine-readability of XML schema has enabled the development of software libraries (e.g. [64]) that are capable of validating and parsing XML files with respect to any user-defined schema. Thus, XML has eliminated the need to create a custom file parser for every application.

Many XML standards have been developed for the exchange of specific types of data within bioinformatics, such as sequence annotations [65, 66], structures [67, 68], alignments [69], protein
1.3. Integration Technologies and Related Research Projects

interactions [70], expression data [71, 72], and pathways [73, 74]. While initially there was a multitude of competing XML formats for each type of data, most areas of bioinformatics appear to be consolidating. For instance, there was once a number of different XML standards for annotating sequences such as SBML, AGAVE, GAME, and BioML; however, EMBL, NCBI, and DDBJ now offer their annotations for download in a common XML format called INSDSeq (International Nucleotide Sequence Database). Another example of consolidation is the fact that several protein interaction databases such as DIP [14], MINT [75], and IntAct [76] now publish their data in a common XML format called PSI-MI [70]. Similarly, the pathway standard SBML [73] is now being utilized by a large number of simulation systems and databases [77]. One important area where standardization has not yet been fully achieved is the sharing of microarray expression data. In spite of ongoing standardization work by the MGED society [78], including the development of the MAGE-ML [71] and MAGE-TAB [79] formats, the three largest microarray repositories [22–24] still publish their datasets and descriptions of experimental conditions in different (although highly similar) text-based formats. Only ArrayExpress is currently using the standards developed by MGED.

The establishment of XML standards is an important step forward for bioinformatics because it greatly facilitates the exchange of data between databases. While database owners might still need to develop scripts for importing and exporting XML, in most cases they will only need worry about mapping their database schema to one or two XML schemas, rather than a myriad of possible formats. However, it is important to note that the existence of XML standards does not address the problem of integrating different types of data. For example, suppose that in addition to the PathwayML file of listing 1.1, a researcher also has a TargetML file containing data about putative drug targets, as shown in Fig. 1.1. The integration of XML schemas is no easier to automate than the integration of relational schemas [80], and thus in order to create an XML file that incorporates both datasets, the researcher must design a new XML schema by hand. This barrier to merging the two datasets prevents a researcher from readily answering questions such as “What are the natural substrates of enzymes that are targeted by the drug Pravastatin?”

<?xml version="1.0" encoding="UTF-8"?>
<drug drugbankID="DB00175" xmlns="http://www.drugml.org/schema">
    <drugname>Pravastatin</drugname>
    <target>UniProt:P04035</target>
    <target>UniProt:P01234</target>
</pathway>

<?xml version="1.0" encoding="UTF-8"?>
<pathway pathwayID="KEGG:hsa00234" xmlns="http://www.pathwayml.org/schema">
    ...     <target>UniProt:P04035</target>
    <target>UniProt:P01234</target>
</pathway>

Figure 1.1: An example illustrating the problem of merging two XML datasets. One dataset identifies the targets of the drug Pravastatin (top), while the other dataset gives pathway information (bottom). Although the datasets are related (by the common protein UniProt:P04035), there is no automated procedure for merging them into a single, integrated dataset.
1.3.3 The Semantic Web

The vision of the Semantic Web is the establishment of a global, interconnected network of machine-readable data, analogous to the network of human-readable documents that currently populates the World Wide Web (WWW) \[81\]. The idea was first conceived by the inventor of the WWW (Tim Berners-Lee), and under his direction, a large amount of work has been conducted by the World Wide Web Consortium (W3C) \[82\] in order to develop the requisite standards \[83\]. Although the standards work is largely complete, the Semantic Web is still more a dream than reality; the vast majority of data owners on the web have not yet shown an interest in participating \[84\].

Regardless of its long term success, the Semantic Web effort has provided a number of valuable tools for addressing the data and software integration problems that currently exist in bioinformatics. In this section, we provide an introduction to RDF and OWL, the core standards for the Semantic Web. We highlight the main advantages of the RDF/OWL framework for data integration, in comparison to the prevailing models of relational databases and XML, and describe several bioinformatics projects where the standards are being used.

RDF

RDF (Resource Description Framework) is the data model for the Semantic Web. Under the RDF model, a dataset is encoded as a set of triples, where each triple represents an atomic statement or “fact”. Each triple consists of a subject, a predicate, and an object, and each of these three parts is identified by a URI (Uniform Resource Identifier). URIs are globally unique identifiers for the WWW; in the context of RDF, URIs may represent anything (people, material objects, categories, types of relationships, etc.)\[1\]. For instance, supposing that http://www.genome.jp/R00335 represents a certain reaction, http://www.genome.jp/hasEnzyme represents the reaction-to-enzyme relationship, and http://www.uniprot.org/P04035 represents a certain protein, the following triple asserts that the protein is an enzyme for the reaction:

(kegg:R00335, kegg:hasEnzyme, uniprot:P04035)\[2\]

A set of such triples constitutes an RDF dataset, and can be visualized as a graph where the subjects and objects are nodes, and the predicates are edges, as in Fig. 1.2. Thus, RDF is a graph-based data model, whereas the relational model is table-based, and the XML model is tree-based. RDF datasets are typically serialized as TURTLE \[85\] or RDF/XML \[86\], and may be stored in specialized databases called triple stores \[87\]. Triple stores are typically queried via the SPARQL query language \[88\], which will be introduced in Section 2.2.

At the current time, RDF is not a widely used format for publishing data within bioinformatics \[89\]. In fact, the author is aware of only one data provider that natively publishes RDF: the UniProt protein database \[11\]. However, several projects have endeavoured to provide publically accessible, third-party translations of existing bioinformatics resources in RDF form. The largest and longest-standing effort in this area is the Bio2RDF \[90\] project, which provides “RDFized” versions of popular databases such KEGG \[25\], Entrez Gene \[9\], and PDB \[17\]. More recently, the LODD (Linked Open Drug Data) project \[91\] has produced RDF versions of drug-related resources such as DrugBank \[21\], DailyMed \[92\], and SIDER \[93\]. Bio2RDF and LODD house the RDF versions of each database in a separate triple store; however, there have also been several projects that have built RDF-based data warehouses, such as the Neurocommons Knowledge Base \[59\] (neuroscience resources), the Pathway Knowledge Base \[60\].

---

1. The most widely used form of URIs in RDF are URLs (Uniform Resource Locators, such as http://www.yahoo.com/).
2. These URIs, and the others that are used in the examples of this section are hypothetical. At the current time, the majority of data providers in bioinformatics do not yet publish their data in RDF, and thus they do not have established URI schemes for referencing their database records. UniProt is an exception.
1.3. Integration Technologies and Related Research Projects

@prefix kegg: <http://www.genome.jp/> .
@prefix uniprot: <http://uniprot.org/> .

kegg:hsa00234 kegg:hasReaction kegg:R00335 .
kegg:R00335 kegg:hasEnzyme uniprot:P04035 .
uniprot:P04035 kegg:hasSubstrate pubchem:C3649 .
uniprot:P04035 kegg:hasProduct pubchem:C3708 .

(a) RDF serialized as TURTLE

<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
 xmlns:kegg="http://www.genome.jp/"
 xmlns:uniprot="http://uniprot.org/"
 xmlns:rdf="http://www.w3.org/1999/02/rdf-syntax-ns#">
  <rdf:Description rdf:about="http://www.genome.jp/hsa00234"
    rdf:resource="http://www.genome.jp/R00335"/>
  <rdf:Description rdf:about="http://www.genome.jp/R00335"
    rdf:resource="http://uniprot.org/P04035"/>
  <rdf:Description rdf:about="http://uniprot.org/P04035"
</rdf:RDF>

(b) RDF serialized as RDF/XML

(c) RDF visualized as a graph

Figure 1.2: An example RDF dataset describing part of a metabolic pathway
1.3. Integration Technologies and Related Research Projects

(pathway resources), and BioGateway [94] (general resources such as Uniprot and GO). RDF has proven useful in warehouse construction because it greatly simplifies the merging of datasets from different sources, as will be described below.

The use of globally unique identifiers in RDF provides a mechanism for matching entities across datasets, which is essential for data integration (see “entity matching” in 1.2). However, the manner in which these global identifiers should be assigned to entities is not entirely clear. URIs were chosen to play the role of identifiers in RDF because it is a scheme that guarantees uniqueness and is already well established on the WWW. In addition, its connection to the DNS framework gives a clear mechanism for the ownership of URLs, in which a single party has control of each URL and the data that it resolves to. However, the use of URLs in RDF has also been ongoing source of confusion [89, 95]. For instance, RDF providers have tended to invent their own URLs for every entity that they reference in their data, regardless of whether URLs for those entities already exist elsewhere. This behaviour has arisen from the fact that an HTTP URL can only resolve to one network location, and thus the use of a “foreign URL” to identify an entity (e.g. a protein) will prevent browsing or crawling to the providers own unique data about that entity. An additional problem is that, even in cases where an RDF data provider is willing to use external URL for an entity, he/she has no straightforward method for determining if such an URL exists. These problems (and several others) motivated the recent development of the LSID (Life Science IDentifier) [42] identifier and resolution system. LSID utilizes features of the DNS system to implement resolution to multiple sources, and in addition seeks to guarantee the persistence and immutability of identifiers and their associated data records. However, there has been a great deal of argument about whether specialized resolution mechanisms are truly necessary for the Semantic Web (e.g. [42, 103]), and LSID has not seen widespread acceptance. At the current time, the issues surrounding the creation and resolution of identifiers in RDF remain unresolved.

The principle advantage of RDF as a data model is that it supports an automated procedure for merging data sets. To perform an RDF-merge, one simply adds the set of triples in one dataset to the set of triples in another dataset (discarding duplicates), as depicted in Fig. 1.3. An RDF-merge will yield a coherent dataset provided that:

1. common entities and common relationships are identified by the same URIs, and
2. whenever the same type of data is encoded in both datasets (e.g. reactions), the graph structures used for representing that data are the same.

For instance, Fig. 1.4 shows a scenario where different graph structures are used to encode biochemical reactions. The merged dataset is not coherent and cannot be uniformly queried to extract information. The problem illustrated in Fig. 1.4 is a variant of the schema integration problem, which also exists under the relational model and the XML model. On the Semantic Web, the problem is addressed through the use of OWL ontologies, as will be discussed in the following section. Although RDF does not provide a universal solution to the schema integration problem, it is important to note that RDF does provide a solution when the domains of the two datasets differ and common identifiers are used, as in Fig. 1.3.

**OWL**

OWL (Web Ontology Language) is a standard for defining ontologies for use with the Semantic Web. An ontology is often defined as a “formal conceptualization of a domain” [96]. In this context, a “conceptualization” is a set of classes of things (e.g. proteins, genes, chromosomes), together with the relationships that connect them (e.g. a protein is encoded by a gene); a “domain” is any possible area of study (e.g. tertiary structures of proteins). OWL ontologies on the Semantic Web serve two purposes:
1.3. Integration Technologies and Related Research Projects

Figure 1.3: An example of a successful RDF-merge.
1.3. Integration Technologies and Related Research Projects

Figure 1.4: An example of an unsuccessful RDF-merge. The two original datasets represent data reactions using different graph structures. uniprot:P04035 and uniprot:P07096 represent two alternate enzymes that catalyze the same reaction.
1.3. Integration Technologies and Related Research Projects

1. They provide the vocabulary and structural rules for encoding an RDF dataset for a given domain. In this respect, OWL ontologies are the functional analog of schemas for relational databases and XML.

2. They encode knowledge about a domain in machine-readable form, such that it can be used to make automated inferences about RDF data. For example, an OWL reasoner might infer that an instance of the class Ligase has at least one substrate and at least one product, given that the class Ligase is a subclass of Enzyme.

An OWL ontology represents a set of axioms in a description logic \[97\], and so it is necessary to understand the basic principles of description logics in order to understand the manner in which OWL ontologies achieve the two tasks above. Every description logic represents four kinds of elements:

**Individuals** Individuals are the entities that are described and related within the data. In the RDF/OWL framework, individuals are represented by the subject and object URIs of the triples in an RDF dataset.

**Properties** Properties are relationships between individuals, and are also referred to as predicates or roles. An OWL ontology provides a vocabulary of predicate URIs that may be used when encoding RDF triples in a given domain; for instance, a pathway ontology corresponding to the dataset in Fig. 1.3 would contain the predicate URIs kegg:hasReaction, kegg:hasEnzyme, etc. In addition, the pathway ontology could assert axioms about predicates such as: “the subject URI of any triple with the kegg:hasReaction predicate belongs to the class Pathway”:

\[
\text{Pathway}(x) \iff \text{hasReaction}(x, y)
\]

In a description logic, classes are represented as unary predicates (e.g. \(\text{Pathway}(x)\)) and properties are represented as binary predicates (e.g. \(\text{hasReaction}(x, y)\)). (Classes will be discussed below.) Many other types of axioms regarding predicates are also possible, such as the assertion that one property is the inverse of another:

\[
\text{isReactionOf}(y, x) \iff \text{hasReaction}(x, y)
\]

**Assertions** Assertions are statements about individuals. In the RDF/OWL framework, assertions are represented by the triples of an RDF dataset.

**Classes** Classes are sets of individuals, and are also referred to as concepts. Classes constitute another part of the vocabulary that is provided by an OWL ontology. The members of a class may be specified explicitly; however, classes may also be defined more generally in terms of axioms. Such axioms describe either necessary or necessary and sufficient conditions regarding individuals that are members of a class. For instance, a class axiom may assert that whenever an individual \(x\) belongs to class \(X\), it also belongs to class \(Y\):

\[
Y(x) \subseteq X(x)
\]

This axiom asserts that \(Y\) is a subclass of class \(X\). More complex relationships between classes can also be encoded in description logics, by building expressions involving the intersection, union, and complement of classes. Classes may also be defined in terms of the properties of an individual. For instance, one might define a class called Enzyme as the set of individuals that have at least one hasSubstrate property and at least one hasProduct property:

\[
\text{Enzyme}(x) \iff \exists y \text{hasSubstrate}(x, y) \land \exists y \text{hasProduct}(x, y)
\]
In OWL, the atoms ∃y hasSubstate(x, y) and ∃y hasProduct(x, y) are called property restrictions. More specifically, they are existential property restrictions. Other types of property restrictions are universal restrictions, which assert that all values of a property must belong to certain class, and cardinality restrictions, which assert that an individual must have a certain number of distinct values for a particular property.

The use of axioms to define classes and properties makes it possible to automatically find contradictions within an ontology; this operation is called consistency checking, and it is one of the principal tasks that can be performed by an OWL reasoner (e.g. [98–100]). It is also possible to use consistency checking to test whether a given RDF dataset conforms to the rules of an ontology; thus, an OWL ontology can be used to specify structural rules of RDF data in a very sophisticated manner. Beyond consistency checking, an OWL reasoner may be used to deduce facts that are not explicitly stated in the data. Typical questions that an OWL reasoner can answer are:

⇒ Are all instances of class X also instances of class Y? (Subsumption)
⇒ Does individual x belong to class X? (Realization)
⇒ Is it possible for any individual to be a member of class X? (Satisfiability)
⇒ Are there any logical contradictions in the ontology? (Consistency)

While this type of “reasoning” may seem somewhat limited, the ability to automatically apply a classification system to biological data has many applications. For instance, an OWL ontology can be used to group proteins into families according to their domains, as described in [101], or to organize a database of small molecules by their functional groups, as in [102].

Description logics are distinguished by the particular constructs (e.g. existential restrictions) that they permit in the construction of their axioms. For example, there are several variants of OWL based on different description logics; the most widely used variant, OWL DL, is a representation of the description logic \( \mathcal{SHOIN}(\mathcal{D}) \). Each letter in indicates one or more constructs:

\( S \) indicates the constructs of the description logic \( \mathcal{ALC} \), which include negation of class expressions, intersections of classes, universal restrictions, and existential quantification.

\( H \) subproperty axioms

\( O \) ordinals (classes may be defined as enumerated lists of individuals)

\( I \) inverse properties

\( N \) cardinality restrictions

\( \mathcal{D} \) datatype properties (properties which take a number or string as their value, rather than an individual)

The set of constructs used by a description logic determines its expressivity. One of the main subjects of research in description logics is the tradeoff between expressivity and computational complexity of reasoning. It is generally understood that the more expressive a logic, the more computationally expensive are its inferencing services (e.g. subsumption), with undecidability as the worst case [103].

Several projects have created OWL ontologies for specific domains of bioinformatics, such as BioPAX [104] for modelling pathways and interactions, OBI [105] for modelling experimental procedures and conditions, and the MGED ontology [78] for modelling microarray experiments. Larger OWL ontologies have also been constructed that cover multiple domains. For instance, the NCI (National Cancer Institute) Thesaurus [106] models many areas of biology and clinical science such as diseases, genes, drugs, pathways, taxonomy, and cancer diagnoses. In addition, a
number of description logic ontologies predate OWL, such as the GALEN \cite{107} and SNOMED-CT \cite{108} ontologies for medical terms, and the TAMBIS ontology \cite{109} for general biological concepts.

It is worth noting here that the word “ontology” is more commonly used in bioinformatics as a synonym for controlled vocabularies (simple hierarchies of terms) such as the GO (Gene Ontology) \cite{110}. GO has been a very successful means of ensuring that scientists use consistent terms when annotating data records, and has enabled the querying of databases using only one standardized term per concept (e.g. “transcription”).

As it relates to data integration, the principal advantage of the RDF/OWL framework over the relational and XML models is that a single dataset can conform to multiple schema (ontologies) at the same time. For example, the dataset of Fig. 1.3 could be queried using two independently developed ontologies for drug targets and enzymes, as shown in Fig. 1.5. This “mixing and matching” of ontologies is possible under OWL because class definitions do not require individuals to have a fixed number of property types; in the example, an instance of the class DrugTarget is permitted to have any number of other properties in addition to “targetOf” (such as “hasSubstrate” and “hasProduct”). Figure 1.5 demonstrates how the RDF/OWL framework solves the schema integration problem when two ontologies describe different types of data about the same entities. However, if the domains of two ontologies overlap (e.g. two ontologies that describe pathways), the representation of common entities must agree (as demonstrated in Fig. 1.4).

Figure 1.5: An example illustrating how different OWL ontologies may be used to provide alternate interpretations of the same RDF dataset. In the first case (left), uniprot:P04035 is identified as an instance of the class Enzyme whereas in the second case (right), uniprot:P04035 is identified as an instance of the class DrugTarget.

\begin{align*}
\text{Enzyme}(x) & \iff \exists y \text{hasSubstrate}(x, y) \land \exists y \text{hasProduct}(x, y) \\
\text{DrugTarget}(x) & \iff \exists y \text{targetOf}(x, y)
\end{align*}

(a) uniprot:P04035 as an instance of the class Enzyme  \hspace{1cm} (b) uniprot:P04035 as an instance of the class DrugTarget

1.3.4 Web Services

The fundamental idea underlying web services is to make the functionality of software accessible over a network. This can be accomplished by setting up a server that listens for input messages from remote clients, performs computations on those input messages, and responds with output messages that contain the results. The client-server model is already widely used for a large number of internet applications such as web servers and FTP servers; the idea is merely to
extend the pattern to other types of software. It important to note that the intended “users” of web services are not people but rather machines; web services are a mechanism for accessing external software from within a program that is analogous to a procedure call. Thus, there is a strong emphasis on machine-readability and automation.

A number of standards have been developed to promote uniform interaction, description, and discovery of web services, which together form the WS-* stack. First, SOAP (Simple Object Access Protocol) \[111\] is a simple XML “wrapper” that allows control information to be attached to web service messages, for purposes such as application-specific routing and security. Second, WSDL (Web Service Description Language) \[112\] is an XML standard for describing the message formats, calling pattern, location, and protocol (e.g. HTTP) for a web service. WSDL files describe message formats using XML schema. Lastly, UDDI (Universal Description, Discovery and Integration) \[113\] is a distributed, XML-based registry system for web services, which can itself be queried as a web service. Apart from the WS-* stack, an alternative model for web services also exists in the literature called RESTful services. Requests to RESTful services are typically encoded as HTTP GET URLs and are invoked by a simple request-response pattern; in other words, RESTful services are invoked in the same manner that a web browser retrieves a web page. The term REST (REpresentational State Transfer) describes a general set of design principles for distributed systems that were employed in the development of the HTTP protocol; the essential characteristics of a REST system are the separation of interactions into clients and servers, the statelessness of servers, the use of global identifiers for resources, and the support of a generic interface for retrieving representations of resources \[114\]. Currently, there is an ideological battle being waged between supporters of the RESTful model and the “Big Services” model \[115\]; however, the two views are in many ways compatible, because the WS-* stack supports a superset of the behaviour described by RESTful services.

The majority of web services in bioinformatics communicate via SOAP, are described in WSDL, and behave in a RESTful manner (in the sense that they are stateless, have a simple request-response calling pattern, and operate over HTTP). Examples include EBI’s dbfetch \[116\] service, the KEGG API \[117\], BioMoby \[118\], and BioMart’s “MartServices” \[119\]. While the WSDL and SOAP standards are widely used in bioinformatics, the UDDI standard is not; in current practice, bioinformaticians usually discover web services through papers, internet search engines, and word of mouth. In an effort to fill this gap, a number of projects have created specialized registries for bioinformatics services with capabilities beyond that of UDDI. For example, the BioMoby \[118\] registry includes a shared ontology of XML datatypes that represent common entities such as DNA sequences and GO terms. Services that wish to participate in the BioMoby framework must consume and generate one of the BioMoby datatypes (or extend the shared ontology), and thus the framework encourages the creation of interoperable services. The Feta \[120\] registry takes an alternate, annotation-based approach. Feta service providers annotate services and their parameters with terms from the myGrid ontology \[121\], enabling users to discover services based on the types of parameters they consume/generate (e.g. “Protein Sequence”), and the types of analyses they perform (e.g. “Sequence Alignment”). Most recently, the BioCatalogue \[122\] provides an open web-based service registry which can be searched by keyword, service categories (e.g. “Phylogeny”), and user annotated tags. The BioCatalogue now contains more than 1000 services.

Beyond the basic tasks of locating and invoking individual services, it is also possible to chain services into workflows to perform complex analyses. (Fig. 1.6 depicts an example workflow, constructed in Taverna \[123\].) Workflows are useful for a number of tasks in bioinformatics. For example, one might build a workflow to automate the mapping of differentially expressed genes to pathways \[124\], the retrieval of gene information for a QTL \[125\], or the construction of a phylogenetic tree \[126\]. A number of software tools have been developed to aid in the (manual) construction of workflows, such as Taverna \[123\] and Kepler \[127\]. Unfortunately, building workflows is still not easy. Chaining services from different providers tends to be difficult and time-consuming \[128\] for the same reason that merging data from different providers is dif-
Figure 1.6: Example Taverna workflow which generates a BLAST report, multiple sequence alignment, and phylogenetic tree for an input protein sequence.
1.3. Integration Technologies and Related Research Projects

Facult: the schema integration problem. Virtually all web services utilize XML-based message formats, and the XML schema for these formats are developed independently by different service providers. Thus, at the current time, building analysis pipelines with web services has few advantages over building pipelines with locally installed software. One advantage that web services do have over ordinary software is platform independence; a web service may be invoked from virtually any programming language on any operating system, and requires no downloading, installation, or configuration prior to use. (The price of this convenience is dependency on a remote server.) In addition, web services are a useful mechanism for “wrapping” a related set of databases and software packages, so that they are exposed with a uniform set of interfaces.

A number of projects have been dedicated to building more sophisticated infrastructure on top of the existing web service standards. For example, grid architectures seek to unify a collection of distributed servers and allow them to be used as a single computational resource. Thus, the functionality that is supplied by a grid system is analogous to the functionality supplied by an operating system: job control (running, pausing, or killing a program), job monitoring, file storage/transfer, and security. In contrast to users of stand-alone web services, grid users must supply both the input data and the software to be run as a job on the grid. The general architecture and functionality of a grid system is described in the OGSA (Open Grid Services Architecture) Architecture document [129], and the de facto standard implementation of OGSA is the Globus Toolkit [130]. Both the BIRN (Bioinformatics Research Network) [131] and caGrid [132] grid projects are based on the Globus Toolkit, as well a number of other grid projects outside of bioinformatics. A related term, cloud computing, is used to describe grid-like services that have recently become available from Amazon [133], Google [134], and Microsoft [135]; these services allow application developers to run multiple instances of an application on virtual machines that are supplied by the vendor. The main feature of cloud computing services is that the number of virtual machines running at any given time may be readily increased or decreased (in some cases automatically). The most common target application of cloud computing is the creation of websites that can readily scale in response to increasing popularity.

Another area of research related to web services is Semantic Web Services [136], which seeks to apply Semantic Web standards to the problem of automatically constructing workflows. The general approach taken in this field is borrowed from the planning domain [137] of artificial intelligence, in which the condition of the world before and after any action is modeled by a set of state variables. The user’s goal in executing the workflow is expressed as a desired set of values for the state variables, and the job of a service broker (e.g. [138]) is to find a path from the initial conditions, through a number of web service invocations, to the goal. Two standards-in-progress have emerged for describing web services along these lines, called OWL-S [139] and WSMO[140].

1.3.5 Mediator Systems

A mediator system (also called a multidatabase system or a data integration system) provides a common query interface to a collection of data sources that are distributed across a network. A mediator answers a query by translating it into a set of subqueries that are issued against the individual data sources (Fig. 1.7). Typically, users formulate their queries against a “virtual” schema or ontology that provides a coherent view of all available data; thus, designing the virtual schema, and mapping it to the schemas of the individual sources, is one of the central tasks of implementing a mediator system. This design/mapping work is yet another instance of the schema integration problem that has been encountered under several different scenarios in this chapter. As in the previous instances, the mappings must be made manually after careful study of the sources. One common technique for distributing the work of schema integration is for each participant to implement a wrapper interface (i.e. a web service) over his/her data source, as depicted in Fig. 1.7. The wrapper layer also masks other types of heterogeneity, such as the different interfaces used for invoking an analytical program and querying a database.
1.3. Integration Technologies and Related Research Projects

A number of mediator systems have been developed for bioinformatics. One of the earliest systems was Kleisli [141]. Kleisli uses a query language called CPL (Collections Programming Language) [142] that allows for the querying and transformation of nested data structures consisting of lists and sets. This is particularly suitable for bioinformatics resources, which tend to have complex structures, such as the sequence features (e.g., exon regions within a DNA sequence) that are embedded in a GenBank record. In the late 90’s, a library of Kleisli “drivers” (i.e., wrappers) was built [143] that enabled querying across a number of important bioinformatics resources such as GenBank, GDB, and BLAST. One caveat of Kleisli is that it requires the user to be aware of the complete set of available data sources, and the particular record structures of each; in other words, there is no virtual schema. The TAMBIS [144] project addressed this issue by mapping Kleisli drivers to the concepts and relationships of a large description logic ontology [109] for bioinformatics. In addition, TAMBIS provided an intuitive graphical user interface for composing queries against the ontology. (The TAMBIS project is no longer active.)

BioFlow [145] is another bioinformatics mediator system, which implements extensions to SQL syntax that allow for the mapping of remote XML files to tables, and the mapping of web forms to SQL functions.

The mediator approach to data integration is often contrasted with the data warehousing approach. On one hand, mediators are not affected by the “staleness” issue of data warehouses, because they obtain their data directly from the original sources. On the other hand, interaction with the original sources often requires large data transfers across the network, and thus mediator queries are generally slower than warehouse queries. In addition, mediators depend on the availability and performance of remote servers, which may be unpredictable. Both the warehouse approach and the mediator approaches typically require the creation of complex schemas and schema mapping rules, which entail a significant investment of human labour. In addition, ongoing maintenance of the schema and associated mappings is required, as new data sources

---

3CPL is similar in syntax to Javascript Object Notation (JSON).
are added and the schemas of existing sources change. One advantage of the mediator approach with respect to maintenance is that the work can be distributed across data providers, if each provider is responsible for maintaining the wrapper of his/her own resource.

1.4 Thesis Outline

This thesis describes the design and implementation of SHARE (Semantic Health and Research Environment), a general-purpose mediator system for bioinformatics. The objective of SHARE is to provide a framework that allows for flexible and efficient querying across distributed databases and software, while at the same time allowing for the ongoing addition of resources by third parties. The principal challenges of implementing a successful mediator system relate to schema integration and query performance. In the following chapters, we propose new approaches for addressing these problems based on the use of Semantic Web technologies. In Chapter 2, we describe the system architecture for SHARE, with an emphasis on the design decisions that support open addition of resources, while avoiding the requirement for centralized development and maintenance of an overarching schema. In Chapter 3, we investigate the problem of optimizing queries in SHARE. We describe an adaptive query execution algorithm called GREEDY, which has been designed to cope with the problem of planning queries in the absence of statistics about the data sources. Chapter 4 concludes by highlighting the contributions of SHARE to bioinformatics and the Semantic Web, and also the weaknesses of the system and areas for future work.
Chapter 2
SHARE System Architecture

2.1 Introduction

SHARE (Semantic Health and Research Environment) is a mediator system that enables simultaneous querying of databases and analytical programs that are distributed across the web. SHARE differs from other mediators, both inside and outside of bioinformatics, because its design is based entirely on Semantic Web standards. All services (i.e. data sources) consume and generate RDF, input/output datatypes are matched using OWL reasoning, and user queries are expressed in SPARQL, the standard query language for RDF. In this chapter, we will demonstrate how these standards can be employed to simplify the implementation, maintenance, and addition of data sources to a mediator system. In particular, there are two characteristics that are unique to SHARE:

1. There is no master schema.
2. If there is a chain of service calls that will convert datatype A to datatype B, that chain can be discovered and invoked automatically.

We will return to these two claims and discuss their implications after describing the mechanics of the system. We begin by introducing SPARQL, the language for SHARE user queries, in Section 2.2. Section 2.3 describes the different types of data sources that are queried by SHARE, and Section 2.4 explains the process of query resolution. Section 2.5 describes the online demonstration of SHARE, and Section 2.6 concludes by discussing the main strengths and weaknesses in the design of the system.

2.2 The SPARQL Query Language

A SPARQL SELECT query searches an RDF dataset for a subgraph with a specified structure, as demonstrated by the example query in Fig. 2.2. The subgraph of interest is defined by a set of triple patterns in the WHERE clause, in the same manner that an RDF graph is described by a set of triples. The difference between an RDF triple and a triple pattern is that a triple pattern may have a variable (denoted by a '?' prefix) in any combination of its three positions. Variables denote placeholders for unknown values, and the result of a SPARQL query is a table of bindings for the variables. A set of triple patterns forms a basic graph pattern, which may be combined with the UNION (logical OR) and OPTIONAL operators to build

<table>
<thead>
<tr>
<th>?structure</th>
<th>?ligand</th>
<th>?name</th>
</tr>
</thead>
<tbody>
<tr>
<td>pdb:2DN1</td>
<td>pdb:OXY</td>
<td>“OXYGEN MOLECULE”</td>
</tr>
<tr>
<td>pdb:2DN1</td>
<td>pdb:MBN</td>
<td>“TOLUENE”</td>
</tr>
</tbody>
</table>

Table 2.1: The results for the SELECT query of Fig. 2.2 when issued against the dataset in Fig. 2.1
2.2. The SPARQL Query Language

PREFIX : <http://sadiframework.org/ontologies/predicates.owl#> .

uniprot:P00214 :has3DStructure pdb:1FRI .
uniprot:P68871 :has3DStructure pdb:2DN1 .
pdb:F3S :hasLigand pdb:OXY .
pdb:2DN1 :hasLigand pdb:MBN .
pdb:MBN :hasChemicalName "TOLUENE" .

(a) target dataset (N3 format)

uniprot:P00214
:has3DStructure  pdb:1FRI .
uniprot:P68871
:has3DStructure	pdb:2DN1 .
pdb:F3S
:hasLigand	pdb:OXY .
pdb:2DN1
:hasLigand	pdb:MBN .
pdb:MBN
:hasChemicalName "TOLUENE" .

(b) target dataset (graph visualization)

Figure 2.1: An example RDF dataset, shown in N3 and graph form.

PREFIX : <http://sadiframework.org/ontologies/predicates.owl#> .

SELECT ?structure ?ligand ?name
WHERE {  
    uniprot:P68871 :has3DStructure ?structure .
    ?structure :hasLigand ?ligand .
    ?ligand :hasChemicalName ?name .
}

Figure 2.2: A SPARQL SELECT query which asks “What are the ligand(s) of Hemoglobin subunit beta (UniProt protein P68871)?”. The results for the query, when issued against the dataset in Fig. 2.1, are shown in Table 2.1.
2.2. The SPARQL Query Language

PREFIX : <http://sadiframework.org/ontologies/predictates.owl#>
PREFIX uniprot: <http://lsrn.org/UniProt:>

CONSTRUCT {  
  uniprot:P68871 :hasLigand ?ligand .  
  ?ligand :hasChemicalName ?name .  
}  
WHERE {  
  uniprot:P68871 :has3DStructure ?structure .  
  ?structure :hasLigand ?ligand .  
  ?ligand :hasChemicalName ?name .  
}

(a) example CONSTRUCT query

(b) result graph for CONSTRUCT query

Figure 2.3: A SPARQL CONSTRUCT query which tells the system to “Construct an RDF graph in which Hemoglobin subunit beta is directly annotated by its ligands.” The resulting RDF graph for this query, when issued against the dataset in Fig. 2.1, is depicted in part (b).
up complex query expressions. The PREFIX syntax at the top of Fig. 2.2 is used to declare abbreviations for URIs that are used in the query.

A SPARQL CONSTRUCT query generates an RDF graph as output, rather than a table of variable bindings. CONSTRUCT queries are useful for performing structural transformations on RDF, as demonstrated by Fig. 2.3. For further details about SPARQL query syntax, see [88]; however, the explanation provided here is sufficient for the remainder of the discussion in this chapter.

2.3 SHARE Data Sources

2.3.1 SADI Services

SADI (Semantic Automated Discovery and Integration) services are stateless web services that are invoked by issuing an HTTP POST request. Both the request and the response consist of a single RDF document. The input RDF document contains one or more input URIs, and may contain any number of statements (i.e. triples) about those URIs; the purpose of these statements is to provide the service with the information that is required to process the inputs. For instance, the example BLAST service in Fig. 2.4 requires one or more amino acid sequences associated with each input protein, in addition to the URIs identifying the proteins themselves. The set of properties (i.e. predicates) that must be supplied for each input URI are encoded by the input OWL class of the SADI service. Each SADI service also has an output OWL class that serves a similar function; it specifies the set of properties that are attached to the input URIs as a result of the service call. In the example service of Fig. 2.3, the input OWL class is the set of nodes having one or more hasAminoAcidSequence properties, and the output class is the set of nodes with one or more hasHomolog properties. While knowledge of the input OWL class is required to successfully invoke a SADI service, knowledge of the output OWL class is required to determine the type of data that the service generates.

4It would be preferable to specify that each input protein has exactly one amino acid sequence. However, standard OWL reasoners can never deduce instances of a class that is defined by an exact cardinality restriction, due to the open world assumption [137].
2.3. SHARE Data Sources

The input and output OWL classes for a SADI service are specified by the service provider as part of the service description, and this description is accessed by performing an HTTP GET on the service URL. The service description is encoded in RDF using the myGrid service ontology, and may include other details such as a natural language description of the service. An example service description is provided in Appendix A. Any third party may register their service with the central SADI registry by submitting the service URL to http://sadiframework.org/registry; the SADI registry then obtains all other necessary information about the service by downloading and processing the service description.

In most respects, the SADI standard is a straightforward application of RDF and OWL to the domain of web service messaging. The key constraint of SADI is that both the input and output OWL classes are defined with respect to the input URIs. As a result, the difference between the input and output classes can be computed to determine the set of generated properties for a service. For example, the service in Fig. 2.4 has one generated property, which is hasHomolog. The SADI registry stores the generated properties for each service, and allows clients to retrieve services based on a property of interest. The discovery of services by individual properties is unique to the SHARE system, and is responsible for its ability to automatically assemble workflows from SPARQL queries, as will be discussed in Section 2.4.

The SADI services that are currently available in the registry generate properties for a number of common identifier types in bioinformatics, such as UniProt [11], GO [110], KEGG [117], and PDB [17]; these identifiers must be expressed as URIs according to the LSRN URI scheme (e.g. http://lsrn.org/UniProt:P68871). The generated properties for each identifier type are depicted in the predicates diagram at http://biordf.net/cardioSHARE/predicates.html (Fig. 2.5), which represents the schema for querying the system. A user may extend the pool of queriable services/properties by registering his or her own SADI service at http://sadiframework.org. Provided that the service has been implemented correctly according to the SADI standard, the service will be immediately available for use during query resolution. Instructions for building SADI services are available at http://sadiframework.org.

2.3.2 SPARQL Endpoints

SPARQL endpoints are HTTP forms that accept SPARQL queries as input. Virtually all triple stores (e.g. Jena [148], Sesame [149], Virtuoso [150]) can be configured to act as SPARQL endpoints, and there are a growing number of SPARQL endpoints on the web that provide access to RDF versions of biological databases. At the current time, these endpoints are not hosted by the original data providers but are rather supplied by third parties such as Bio2RDF [90] and the Linked Open Drug Data (LODD) [91] project. Bio2RDF provides RDF mirrors of general resources such as the UniProt [11] protein database and the KEGG [117] pathway database, while LODD provides mirrors of drug-related resources such as DrugBank [21] and the SIDER database of side effects [93].

The SHARE query engine is capable of querying across SPARQL endpoints. SHARE implements this functionality by means of a mediator-side service wrapper for each SPARQL endpoint. Viewed from the query engine side, each service wrapper makes a SPARQL endpoint appear as if it is a SADI service; just as for a SADI service, each SPARQL endpoint has a set of generated properties and may be invoked using one or more URIs as input. On the network side, the service wrapper translates the input URIs to CONSTRUCT queries and sends them to the relevant endpoint, as depicted in Fig. 2.6.

The set of generated properties for a SPARQL endpoint is the set of predicates that are occur in one or more triples in the endpoint. In principle, this list of predicates can be obtained automatically by issuing the following query to the endpoint:

```sql
SELECT DISTINCT ?p WHERE { ?s ?p ?o }
```
Figure 2.5: The generated properties for the SADI services that are currently available through the public registry [http://sadiframework.org/registry](http://sadiframework.org/registry). Currently, this diagram is maintained by hand, but in future work it will be constructed automatically from the contents of the registry.
However, this is a relatively expensive query, because it requires iterating through all of the triples in the database. In practice, it tends to fail for large triple stores because the HTTP POST request that issues the query expires before the database finishes processing the query. A further problem is that SPARQL endpoints may disallow the execution of queries that are deemed to be too expensive. At the current time, such endpoints are instead indexed based on a single URI representing a “typical” database record (e.g. http://bio2rdf.org/uniprot:P05067): in order to accommodate complex record structures, the RDF graph rooted at the URI is crawled using a breadth first search. Clearly, this is not an ideal solution, because it is not fully automatic and is not guaranteed to be complete. In future work, the author plans to develop scripts for different types of SPARQL endpoints (e.g. Virtuoso [150], D2R [151]) that will build standardized indices on the endpoint side. In addition to a list generated properties, SHARE’s indices for SPARQL endpoints also contain regular expressions for the subject/object URIs that occur in the endpoint. As with generated properties, the regular expressions can be built by querying the endpoints, but for large endpoints must be constructed using the record-based approach.

The SPARQL endpoint registry for SHARE currently contains indices for the LODD and Bio2RDF endpoints, as listed in Table 2.2. These indices are rebuilt on a weekly basis. A map of the generated properties for the endpoints is shown in Fig. 2.7 as the map is very large, only a small portion is shown in detail here. The map was generated by performing a series of breadth first traversals across the data in the endpoints, with a depth limit of 7 edges for each traversal. The root URIs for the traversals were chosen by querying for a list of distinct record types (i.e. rdf:types) from each endpoint, and then randomly sampling 3 URIs of each type. As with the SPARQL indexes, the map is not guaranteed to be complete; improved methods for constructing the map will be a subject of future work.

---

To view the map in full detail, files may be downloaded in N3, Cytoscape, SVG, and PNG formats from [http://sadiframework.org/sparqlmap](http://sadiframework.org/sparqlmap). Viewing the map with Cytoscape or Inkscape (SVG viewer) is recommended, as PNG requires too much memory to be loaded in most image viewers.
2.3. SHARE Data Sources

<table>
<thead>
<tr>
<th>SPARQL Endpoint URL</th>
<th>Provider</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://ww4.wiwiss.fu-berlin.de/dailymed/sparql">http://ww4.wiwiss.fu-berlin.de/dailymed/sparql</a></td>
<td>LODD</td>
</tr>
<tr>
<td><a href="http://ww4.wiwiss.fu-berlin.de/diseasome/sparql">http://ww4.wiwiss.fu-berlin.de/diseasome/sparql</a></td>
<td>LODD</td>
</tr>
<tr>
<td><a href="http://ww4.wiwiss.fu-berlin.de/sider/sparql">http://ww4.wiwiss.fu-berlin.de/sider/sparql</a></td>
<td>LODD</td>
</tr>
<tr>
<td><a href="http://ww4.wiwiss.fu-berlin.de/drugbank/sparql">http://ww4.wiwiss.fu-berlin.de/drugbank/sparql</a></td>
<td>LODD</td>
</tr>
<tr>
<td><a href="http://ww4.wiwiss.fu-berlin.de/medicare/sparql">http://ww4.wiwiss.fu-berlin.de/medicare/sparql</a></td>
<td>LODD</td>
</tr>
<tr>
<td><a href="http://ww4.wiwiss.fu-berlin.de/stitch/sparql">http://ww4.wiwiss.fu-berlin.de/stitch/sparql</a></td>
<td>LODD</td>
</tr>
<tr>
<td><a href="http://uniprot.bio2rdf.org/sparql">http://uniprot.bio2rdf.org/sparql</a></td>
<td>Bio2RDF</td>
</tr>
<tr>
<td><a href="http://omim.bio2rdf.org/sparql">http://omim.bio2rdf.org/sparql</a></td>
<td>Bio2RDF</td>
</tr>
<tr>
<td><a href="http://reactome.bio2rdf.org/sparql">http://reactome.bio2rdf.org/sparql</a></td>
<td>Bio2RDF</td>
</tr>
<tr>
<td><a href="http://mesh.bio2rdf.org/sparql">http://mesh.bio2rdf.org/sparql</a></td>
<td>Bio2RDF</td>
</tr>
<tr>
<td><a href="http://ec.biol2rdf.org/sparql">http://ec.biol2rdf.org/sparql</a></td>
<td>Bio2RDF</td>
</tr>
<tr>
<td><a href="http://hgnc.bio2rdf.org/sparql">http://hgnc.bio2rdf.org/sparql</a></td>
<td>Bio2RDF</td>
</tr>
<tr>
<td><a href="http://inoh.bio2rdf.org/sparql">http://inoh.bio2rdf.org/sparql</a></td>
<td>Bio2RDF</td>
</tr>
<tr>
<td><a href="http://mgi.bio2rdf.org/sparql">http://mgi.bio2rdf.org/sparql</a></td>
<td>Bio2RDF</td>
</tr>
<tr>
<td><a href="http://protein.bio2rdf.org/sparql">http://protein.bio2rdf.org/sparql</a></td>
<td>Bio2RDF</td>
</tr>
<tr>
<td><a href="http://unists.bio2rdf.org/sparql">http://unists.bio2rdf.org/sparql</a></td>
<td>Bio2RDF</td>
</tr>
<tr>
<td><a href="http://obo.bio2rdf.org/sparql">http://obo.bio2rdf.org/sparql</a></td>
<td>Bio2RDF</td>
</tr>
<tr>
<td><a href="http://uniparc.bio2rdf.org/sparql">http://uniparc.bio2rdf.org/sparql</a></td>
<td>Bio2RDF</td>
</tr>
<tr>
<td><a href="http://cpd.bio2rdf.org/sparql">http://cpd.bio2rdf.org/sparql</a></td>
<td>Bio2RDF</td>
</tr>
<tr>
<td><a href="http://affymetrix.bio2rdf.org/sparql">http://affymetrix.bio2rdf.org/sparql</a></td>
<td>Bio2RDF</td>
</tr>
<tr>
<td><a href="http://irefindex.bio2rdf.org/sparql">http://irefindex.bio2rdf.org/sparql</a></td>
<td>Bio2RDF</td>
</tr>
<tr>
<td><a href="http://kegg.bio2rdf.org/sparql">http://kegg.bio2rdf.org/sparql</a></td>
<td>Bio2RDF</td>
</tr>
<tr>
<td><a href="http://cpath.bio2rdf.org/sparql">http://cpath.bio2rdf.org/sparql</a></td>
<td>Bio2RDF</td>
</tr>
<tr>
<td><a href="http://biocyc.bio2rdf.org/sparql">http://biocyc.bio2rdf.org/sparql</a></td>
<td>Bio2RDF</td>
</tr>
<tr>
<td><a href="http://chebi.bio2rdf.org/sparql">http://chebi.bio2rdf.org/sparql</a></td>
<td>Bio2RDF</td>
</tr>
<tr>
<td><a href="http://taxonomy.bio2rdf.org/sparql">http://taxonomy.bio2rdf.org/sparql</a></td>
<td>Bio2RDF</td>
</tr>
<tr>
<td><a href="http://homologene.bio2rdf.org/sparql">http://homologene.bio2rdf.org/sparql</a></td>
<td>Bio2RDF</td>
</tr>
<tr>
<td><a href="http://pubchem.bio2rdf.org/sparql">http://pubchem.bio2rdf.org/sparql</a></td>
<td>Bio2RDF</td>
</tr>
<tr>
<td><a href="http://biocarta.bio2rdf.org/sparql">http://biocarta.bio2rdf.org/sparql</a></td>
<td>Bio2RDF</td>
</tr>
</tbody>
</table>

Table 2.2: List of SPARQL endpoints currently indexed by SHARE
2.3. SHARE Data Sources

Figure 2.7: A map of the generated properties for SPARQL endpoints available under SHARE. The callouts show sections of the map containing predicates about chemical reactions, proteins, and drugs. Each node represents a distinct record type (i.e. rdf:type) that occurs in one or more of the endpoints.
2.4 SHARE Query Resolution

2.4.1 Overview

Standard SPARQL query engines \[148, 150\] have limited capabilities for querying remote data \[152\]. Users may include “FROM” clauses at the beginning of a SPARQL query that indicate the URLs of target RDF files, and these files will be downloaded in bulk as the first step of query processing. The main caveat of this approach is that it does not work well for large datasets. In addition, it precludes the possibility of querying data that is generated by analytical software.

SHARE addresses these issues by acting as a transparent data gathering layer on top of a standard SPARQL query engine. An overview of the process is depicted in Fig. 2.8. In the first phase of query resolution, SHARE retrieves the data required to answer the query by issuing a series of requests to available services (SADI services and SPARQL endpoints); the process by which SHARE maps a query to a set of service requests will be explained in detail in the remainder of this section. As data from the services is retrieved, it is aggregated in a local triple store.

In the second phase of query resolution, the user’s original query is run against the local triple store using a standard SPARQL query engine. Conceptually, SHARE provides access to a large dataset that consists of the output from each SADI service for each possible input URI, together with all triples stored by all known SPARQL endpoints. Hereafter, we will refer to this dataset as the virtual graph. The objective of SHARE is to download a subset of the virtual graph that is sufficient to generate a complete solution for the user’s query. In this context, “complete” means that the solution set has all of the solutions that would be found by running the user’s query on a complete, materialized version of the virtual graph.

2.4.2 Resolution Process of an Example SHARE Query

```sparql
1: PREFIX sadi: <http://sadjframework.org/ontologies/properties.owl#>
2: PREFIX uniprot: <http://lsrn.org/Uniprot:>
3:
4: SELECT *
5: WHERE {
8: }
```

Listing 2.1: An example SHARE query which asks: “What biological pathways does Xanthine dehydrogenase/oxidase participate in?”

To illustrate the process by which SHARE retrieves and assembles a dataset for a SPARQL query, we will use an example. Consider the SHARE query shown in listing 2.1, which asks: “What biological pathways does Xanthine dehydrogenase/oxidase participate in?” The PREFIX lines 1-2 define abbreviations for URIs that make the query easier to read. The “SELECT *” line indicates that all variables mentioned in the query (?gene and ?pathway) should be included in the table of solutions. Lines 5-8 define the WHERE clause, which specifies the search criteria for the query. Here, the WHERE clause consists of two triple patterns (lines 6 and 7), which together constitute a single basic graph pattern. The criteria specified by each of the triple patterns are combined with the logical AND operation. Thus, each solution for the query of listing 2.1 is a combination of values for ?gene and ?pathway such that:

1. Xanthine dehydrogenase/oxidase (uniprot:P47989) is encoded by ?gene, AND
Figure 2.8: An overview of SHARE query resolution. SHARE acts as a transparent data gathering layer on top of a standard SPARQL engine. In the first phase of query resolution, SHARE translates the user’s query to a series of service requests, in order to gather the data needed to answer the query. This data is accumulated in a local triple store as it is retrieved. In the second phase of query resolution, the user’s original query is executed against the local triple store using a standard SPARQL query engine.
2.4. SHARE Query Resolution

SHARE builds a dataset for a basic graph pattern by retrieving data for each triple pattern in sequence. Each triple pattern is mapped to a set of service requests, and the output data from those requests is used to determine bindings for any unbound variables that occur in the pattern. To illustrate, we will consider the resolution of the example query in listing 2.1. SHARE resolves the first triple pattern, (uniprot:P47989, sadi:isEncodedBy, ?gene), by the following steps:

1. Matching services are identified. Matching services are services that output triples of the form (<input URI>, sadi:isEncodedBy, *), where <input URI> is an input URI in the service request, and * denotes any possible value. Such services are said to “attach” sadi:isEncodedBy to their input URIs, or (equivalently) to have sadi:isEncodedBy as one of their generated properties.

2. Matching services are invoked. In this case, each service request contains a single input URI: uniprot:P47989. It is possible to invoke services with more than one input URI; this would be necessary if the subject of the triple pattern were a variable with multiple bindings, rather than a constant URI.

3. The output RDF from the services is loaded into the local triple store.

4. Bindings are assigned to any unbound variables in the current triple pattern. SHARE queries the local triple store for triples of the form (uniprot:P47989, sadi:isEncodedBy, *) that were loaded in Item 3. The full set of distinct values for “*” become bindings for the variable ?gene.

After resolving the first triple pattern, SHARE will proceed to gather data for the second triple pattern, (?gene, sadi:isParticipantIn, ?pathway), in a similar manner. However, for the second pattern, the subject is a bound variable (?gene) rather than a constant URI (uniprot:P47989). The bindings for ?gene, as determined by the resolution of the first triple pattern, become the input URIs for the service requests that gather data for the second triple pattern. After performing the service calls for the two patterns, SHARE has aggregated a dataset that is sufficient to answer the user’s query. The original query is then issued against the local triple store using a standard SPARQL query engine, and the solutions are displayed.

2.4.3 Pseudocode for SHARE Query Resolution

In the example query of the preceding section, all triple patterns were resolved by using the bindings (or constant values) of the subjects as input URIs for the service invocations. We will refer to this as resolving the triple pattern in the forward direction (left-to-right). SHARE is also capable of resolving triple patterns in the reverse direction (right-to-left). Given a triple pattern (s, p, o), the system decides which direction to use according to the following rules:

**Case 1: s is a bound variable or a URI.** The triple pattern is resolved in the forward direction. The value(s) for s are used as input URIs to all services that generate property p; if o is an unbound variable, then the set of retrieved values for property p become the bindings of o.

**Case 2: s is an unbound variable and o is a bound variable or a URI.** The triple pattern is resolved in the reverse direction. The value(s) for o are used as input URIs to all services that generate inverse(p), where inverse(p) is the OWL inverse of property p. The set of retrieved values for inverse(p) become the bindings of s.

---

6 Readers may observe that the second pattern is not strictly correct; the participants of metabolic pathways are proteins and metabolites rather than genes. This is an artifact of the way data is modelled in the KEGG database. In KEGG, the assertion that a gene participates in a metabolic pathway means that the gene codes for at least one enzyme that participates in the pathway.
2.4. SHARE Query Resolution

The pseudocode for the data gathering procedure of SHARE is shown in listing 2.2. Note that the above two cases are the only types of triple patterns that SHARE is capable of resolving. In particular, it is not possible for the system to resolve a triple pattern in which the predicate is a variable, or a pattern in which both the subject and the object are unbound variables. The former case is unresolvable because the predicate URI is the basis for discovering relevant services, and the latter case is unresolvable because there is no input to send the services in order to generate data. Solutions for these limitations will be a subject of future work.

2.4.4 Validating Inputs for Services

One aspect of the data gathering process that remains to be explained is the manner in which SHARE confirms that a URI is a valid input for a service. (This check occurs when getValidInputs is called inside of invokeServicesByProperty, in listing 2.2.) In the case of SADI services, a URI is a valid input if and only if it is an instance of the service’s input OWL class, and thus SHARE must use an OWL reasoner to validate an input URI. However, standard OWL reasoners [98–100] are currently limited to reasoning about locally stored data, just as SPARQL engines are limited to querying local data. Thus, SHARE must perform an additional data-gathering procedure to collect properties that are relevant to an OWL class definition. For example, suppose that the following query were issued to SHARE:

\[
\text{SELECT } ?\text{homolog} \\
\text{WHERE} \{ \\
\text{uniprot: P68871 :hasHomolog } ?\text{homolog} . \\
\}\n\]

Further, suppose that the only hasHomolog-generating service available is the BLAST service shown of Fig. 2.4. This service requires that every input URI belongs to the class ProteinWithSequence, and a URI is an instance of ProteinWithSequence if and only if it has at least one hasAminoAcidSequence property. However, at the time the triple pattern (uniprot:P68871, :hasHomolog, ?homolog) is resolved by SHARE, no attempt has been made to gather hasAminoAcidSequence properties for uniprot:P68871. SHARE addresses this problem with a secondary, OWL-based data-gathering procedure. This procedure automatically decomposes the definition for the input OWL class of ProteinWithSequence, determines that the hasAminoAcidSequence property is required for uniprot:P68871, and invokes all services that generate hasAminoAcidSequence with uniprot:P68871 as input. The hasAminoAcidSequence-generating services have their own input OWL classes, which may themselves require additional data; in this manner, the data-gathering process may proceed recursively through an arbitrarily long chain of dependent services. A base case is reached when a service is discovered that consumes bare URIs. In theory, such services would have “owl:Thing” as their input OWL class. (owl:Thing is a special, predefined OWL class that contains all URIs.) In practice, SHARE contains rules to assign bare URIs to OWL classes representing different types of database records, based on their syntax. For example, uniprot:P68871 is represented as http://lsrn.org/UniProt:P68871 and is assigned to the OWL class http://purl.oclc.org/SADI/LSRN/UniProt_Record. The URLs for SHARE are based on the LSRN (Life Sciences Resource Name) URL scheme [153].

The pseudocode for the OWL-based data gathering procedure is shown in listing 2.3. The main work is performed by the gatherDataForTypePattern procedure; the name for this procedure comes from the convention that instances of an OWL class are indicated by triples of the form (<subject URI>, rdf:type, <OWL class URI>). gatherDataForTypePattern is recursive in two ways. First, calling services to gather new data requires resolving the input OWL classes of those services, which may in turn require gathering additional data (as described in the previous paragraph). Second, OWL class definitions may themselves be nested. For example, an AnnotatedProtein might be defined as any URI that has at least one hasAnnotation property.
2.4. SHARE Query Resolution

```java
resolve(query)

    bindings = empty hash table

    // initialize variable bindings
    for each triple pattern t in query
        if (t.subject is not a variable)
            bindings[t.subject] = singletonSet(t.subject)
        if (t.predicate is a variable)
            error "cannot resolve pattern where predicate is a variable"
        if (t.object is not a variable)
            bindings[t.object] = singletonSet(t.object)

    // gather data
    for each triple pattern (s, p, o) in query
        processPattern(s, p, o, bindings)

    // hand over the gathered data to a standard SPARQL engine
    return SPARQLEngine.query(query, localTripleStore)
```

```java
processPattern(s, p, o, bindings)

    subjects = bindings[s]
    objects = bindings[o]

    if (subjects == null && objects == null)
        warn "cannot resolve pattern where both subject and object are unbound variables"
    else if (objects == null)
        invokeServicesByProperty(subjects, p)
    else if (subjects == null)
        invokeServicesByProperty(objects, inverse(p))
    else
        // both subject and object are already bound
        invokeServicesByProperty(subjects, p)

    for each triple t in localTripleStore.getMatchingTriples(bindings[s],
                                                            bindings[p], bindings[o])
        if(subjects == null)
            bindings[s].add(t.subject)
        if(objects == null)
            bindings[o].add(t.object)
```

```java
invokeServicesByProperty(inputURIs, property)

    services = empty set
    services.add(SADIRegistry.getServicesByProperty(property))
    services.add(SPARQLRegistry.getServicesByProperty(property))

    for each service s in services
        validInputURIs = getValidInputs(inputURIs, s)
        resultTriples = s.invoke(validInputURIs)
        localTripleStore.add(resultTriples)
```

Listing 2.2: Pseudocode for the main data gathering procedure of SHARE. `resolve` is the top-level method that answers a SHARE query. `resolve` invokes `processPattern` for each triple pattern in the query, in order to find solutions for unbound variables. `invokeServicesByProperty` is a helper procedure for `processPattern` that invokes services for a particular generated property and set of input URIs.
2.4. SHARE Query Resolution

`getValidInputs(inputURIs, service)`

validInputURIs = empty set

if service is SPARQL endpoint
    for each URI in inputURIs
        if(service.matchesRegEx(URI))
            validInputURIs.add(URI)
else
    // service is a SADI service
    inputClass = service.getInputClass()

    // gather data that is relevant to the
    // definition of inputClass
    s = generateNewVariableName()
    bindings[s] = inputURIs
    gatherDataForTypePattern(s, inputClass, bindings)

    // check each URI against the inputClass
    for each URI in inputURIs
        if(OWLReasoner.getInstance(URI, inputClass))
            validInputURIs.add(URI)

return validInputURIs

`gatherDataForTypePattern(s, inputClass, bindings)`

// OWL classes may have explicitly declared
// parent classes and equivalent classes
for each parent class P of inputClass
    gatherDataForTypePattern(s, P)
for each equivalent class E of inputClass
    gatherDataForTypePattern(s, E)

// OWL classes may be constructed using set operators
if inputClass is a union of classes
    for each class C in the union
        gatherDataForTypePattern(s, C)
else if inputClass is an intersection of classes
    for each class C in the intersection
        gatherDataForTypePattern(s, C)
else if inputClass is the complement of class C
    gatherDataForTypePattern(s, C)
else
    // Base case: inputClass is defined as a set of property restrictions
    for each property restriction r of inputClass
        p = r.getProperty()
        o = generateNewVariableName()
        if r is a value restriction
            bindings[o] = set(r.getValue())
            processPattern(s, p, o, bindings)
        else if r is a class restriction
            bindings[o] = null
            processPattern(s, p, o, bindings)
            gatherDataForTypePattern(o, r.getClass(), bindings)
        else
            // r is a cardinality restriction
            bindings[o] = null
            processPattern(s, p, o, bindings)

Listing 2.3: Pseudocode for validating inputs to services. `gatherDataForTypePattern` is a helper method for `getValidInputs` which gathers all available data that is related to an OWL class definition. `generateNewVariableName` generates a new variable name that hasn’t been in the user’s query or by previous temporary variables.
2.5. SHARE Online Demonstration

with a value (i.e. object URI) from the class GOTerm. (We call this a “class restriction” in the pseudocode of listing 2.3 in OWL, it is also called a “someValuesFrom” restriction.) In this case, the OWL data gathering procedure will not only have to retrieve hasAnnotation properties for the subject URI, but also properties for the values of hasAnnotation that are relevant to the definition of GOTerm.

In addition to validating inputs for services, gatherDataForTypePattern is also used by SHARE to directly find instances of OWL classes from within a query. For example, the query

```
SELECT ?kinase
WHERE {
    ?kinase rdfs:comment ?comment.
}
```

will identify instances of the class Kinase, provided that the property restrictions used to define myOntology:Kinase map to SADI services.

2.5 SHARE Online Demonstration

The SHARE demonstration site allows users to issue queries against a sample set of SADI services and SPARQL endpoints, and may be accessed at [http://biordf.net/cardioSHARE/query](http://biordf.net/cardioSHARE/query). The main page provides a form for entering SPARQL queries, and a list of example queries is provided at [http://biordf.net/cardioSHARE/queries.html](http://biordf.net/cardioSHARE/queries.html). Fig. 2.9 shows the result of executing the SPARQL query from listing 2.1.

The SPARQL endpoints provided by the Bio2RDF [90] and Linked Open Drug Data (LODD) [91] projects are also queriable by the system. Bio2RDF provides SPARQL mirrors for popular databases such as UniProt [11], KEGG [117], and Pfam [154], while LODD provides endpoints for drug-related resources such as Drugbank [21] and SIDER [93]. At the current time, the generated properties of these endpoints is not incorporated in the predicate diagram of listing 2.2 however, it is feasible to generate a complete diagram automatically from the SPARQL and SADI service registries, with further work.

2.6 Source Code

The source code for the SHARE query engine may be accessed at [http://sadi.googlecode.com](http://sadi.googlecode.com), under the sadi.share folder (i.e. [http://sadi.googlecode.com/svn/trunk/sadi.share](http://sadi.googlecode.com/svn/trunk/sadi.share)). The real methods corresponding to the pseudocode methods of listings 2.2 and 2.3 may be found within the SHAREKnowledgeBase class (ca.wilkinsonlab.sadi.share.SHAREKnowledgeBase), as per Table 2.3. The source code for the demonstration servlet and website is available separately from [http://cardioshare.googlecode.com](http://cardioshare.googlecode.com).

2.7 Discussion

At the current time, XML is the de facto standard for encoding data files and for interacting with web services. Thus, the main caveat of SHARE as an RDF/OWL-based system is that it is not compatible with the majority of existing data and services on the web. However, the RDF/OWL data model has significant advantages over XML, with respect to the implementation and maintenance of a mediator system. In particular, it is possible to automate several tasks that traditionally require hand-coded schema mappings:

**building/maintaining a master schema** The query schema for SHARE (Figs. 2.5 and 2.7) is determined by the generated properties of the available SADI services and SPARQL
Figure 2.9: A screenshot of the SHARE demonstration site, after running the query from 2.1, which asks: “What biological pathways does Xanthine dehydrogenase/oxidase participate in?”

Table 2.3: The methods in the SHARE source code which correspond to the pseudocode methods of listings 2.2 and 2.3. All relevant methods members of the SHAREKnowledgeBase class (ca.wilkinsonlab.sadi.share.SHAREKnowledgeBase), which is located with the sadi.share folder at http://sadi.googlecode.com
2.7. Discussion

endpoints; there is no centralized control of the schema. This is a significant advantage
because the design of a master schema, and the maintenance of mappings to and from
that schema, is one of the largest sources of manual labour involved in implementing
a mediator system. A further advantage is that new SADI services may be added at
any time, and will immediately become queriable by the system. The main drawback of
SHARE's laissez faire approach is that it will likely to result in a more chaotic schema
than that of a centrally managed system. In particular, it will be essential for service
providers to reuse property URIs when describing the same relationships. For instance,
if three services that retrieve DNA sequences generate the properties hasDNASequence,
hasSequence, and hasNucleotideSequence, then a query referencing hasDNASequence will
only invoke one of those services. Moreover, a schema containing many synonymous pred-
icates will be unnecessarily difficult to navigate and understand. Achieving community
agreement on predicates is an important problem not only for SHARE, but also for the
Semantic Web in general. While several basic RDF vocabularies such as the Dublin Core
Metadata Initiative (document metadata) [155], SKOS (taxonomies) [156], and FOAF (re-
relationships between people) [157] have now gained widespread acceptance, it remains to
be seen whether similar standardization can be achieved in more complex domains such as
bioinformatics.

integrating output data from multiple services During SHARE query execution, the
output of each SADI service is automatically integrated into the local triple store by an
RDF-merge operation. In contrast, integrating the results of an XML-based service into
a relational database requires a hand-coded mapping of the output XML to the database
schema.

matching service input/output datatypes Under the current paradigm of XML-based web
services, connecting the output of a service A to the input of a service B requires that
the output XML schema of A is equal to the input XML schema of B. However, there
are many scenarios where the schema are not equal, despite the fact that the two schemas
describe the same type of entity, and the connection itself would be logical. Consider a
scenario in which a bioinformatician wants to connect the output of an XML-based BLAST
service to the input of an XML-based multiple sequence alignment (MSA) service. The
BLAST output is a collection of sequences and the MSA input is likewise a collection
of sequences. However, the precise information provided/required about each sequence
may differ between the two services. For example, the MSA might require the source
organism of each input sequence in order to improve the quality of the alignment, while
this information might not be included in the output of the BLAST service. In this case,
the two services are not directly compatible, and the bioinformatician must construct
an extra “shim” [128] step to translate between the two schemas. This type of problem
arises because XML schema are atomic, and there is no mechanism by which they can be
automatically decomposed or merged. In contrast, SHARE’s use of the RDF data model
allows it to automatically assemble input datatypes from individual properties, as is done
in the gatherDataForTypePattern method of listing 2.3.

There are a number of areas where future work will be needed, in order to make SHARE
a practical tool for bioinformaticians. First, there will need to be a mechanism to inspect the
precise chain of sources and assertions that have led to each query solution (provenance tracking).
This functionality is particularly important given the ability of any third party to add services
to the system, and thus the potential for sabotage by services that make false or non-sensical
assertions. Second, there will need to be controls for data source selection, algorithm selection,
and algorithm parameters (e.g. gap penalty). It is difficult to incorporate such controls into the
query language itself, as it would destroy the abstraction of property-based service discovery,
and greatly would complicate the query interface. However, SHARE could in principle be used
to generate a template workflow (e.g. a SCUFL \cite{158} file) to be edited in workflow software such as Taverna \cite{123}. Although not fully automated, this approach would eliminate a large part of the manual work involved in identifying and wiring together relevant services for an analysis.
Chapter 3

An Adaptive Query Evaluation Algorithm for SHARE

3.1 Introduction

One of the main challenges in implementing a distributed query system is in determining efficient execution plans for queries. The database operations that are used to answer a query (e.g. selections, projections, joins) can often be rearranged into a large number of logically equivalent plans, and these plans can differ greatly in their overall efficiency. This same problem is encountered when evaluating queries against standalone relational databases, and has been studied extensively in this context. However, in a distributed scenario, the optimization problem is more difficult for a number of reasons. First, estimating the cost of a query plan is more complex. Whereas traditional query optimization is based on minimizing the number of I/O operations (i.e. hard drive reads), the time required to process requests at remote servers and the time required to transfer data across the network are also significant factors for distributed systems. Another challenge is that the query system may have little or no statistics about the data that is available from remote sources, while in standalone databases, such statistics are essential for constructing efficient plans. In addition, for many mediator systems such as SHARE, the data sources may be software services (e.g. BLAST) for which there is no standard means of computing statistics.

In this chapter we describe an algorithm called GREEDY that is used by SHARE to evaluate distributed SPARQL queries in an efficient manner. More specifically, the goal of GREEDY is to optimize the ordering of joins across distributed RDF data services. GREEDY differs from the traditional approach to query optimization because the full execution plan for a query is not determined prior to running the query. Instead, GREEDY performs one join at a time, and uses the cardinalities of previously completed joins to decide which join to perform next. Thus, GREEDY is an adaptive optimization algorithm. A further difference between GREEDY and the standard approach to query optimization is that statistics about the data are learned during the execution of queries, which are then used to help the optimizer in the planning of future queries.

We begin by briefly discussing related work in Section 3.2. Next, we review the procedure used by SHARE to resolve queries in Section 3.3 in order to provide context for the optimization problem. In Section 3.4, we present the optimization problem by means of an example. In Section 3.5, we describe a simple optimization relating to the assignment of variable bindings, and in Section 3.6 we present the main optimization algorithm, GREEDY. We evaluate the performance of GREEDY using a small set of bioinformatics queries from the literature in Section 3.7, and we discuss areas for future work in Section 3.8. Finally, we summarize the strengths and weaknesses of GREEDY in Section 3.9 and provide pointers to the relevant source code in Section 3.10.

\footnote{For an overview of the standard approach to query optimization, see [159] and [160].}
3.2 Related Work

GREEDY is an extension of an idea published in [161], in “Section 7: Adaptive query execution using Prim’s algorithm”. A key difference between the algorithm described here and the original pseudocode is that multiple inputs to the same service are batched into a single request. This simplifies the pseudocode for the algorithm, and is also more efficient. Another improvement over the original paper is that specific methods have been developed for gathering and computing predicate statistics.

Much work has been done on distributed query processing in the context of relational databases; for a good overview, see [162]. Some of the main techniques that are used in distributed query processing are dynamic programming (for enumeration of query plans), row blocking, and semi-joins. Applying the dynamic programming approach to a mediator system requires that the wrappers expose methods for estimating the costs of queries to the individual sources. In addition, the enumeration of plans is typically exhaustive, and thus quickly becomes expensive for complex queries. For these reasons, the dynamic programming approach was deemed unsuitable for SHARE, and the GREEDY algorithm was developed instead. The technique of row blocking simply involves sending the rows of table across a network in blocks, rather than individually, when performing a join across sites. In SHARE, this roughly corresponds to the batching of inputs to SADI services. The semi-join is a technique for reducing data transfer when joining two tables at different sites. Rather than transferring an entire table from one site to another, a semi-join first sends a reduced version of Table A to Site B, which contains only the join columns. Site B then performs a join against the reduced table to determine the matching rows in Table B. Finally, the matching rows of Table B are transferred to Site A, and a join is performed between Table A and the matching rows of Table B. SHARE uses a technique similar to a semi-join when performing joins, where the values of the join column correspond to the bindings of a SPARQL variable.

Work has also been done for distributed query optimization in the context of RDF and SPARQL specifically. Other distributed SPARQL systems that have been described in the literature are DARQ [152], SemWIQ [163], and the Distributed SAIL Extension [164] for Sesame [149]. The DARQ paper describes a procedure for query planning that is based on the average selectivities of predicates when resolved in either the forward (bound subject) or reverse (bound object) directions. An important caveat of the DARQ approach is that the selectivity values for the various predicates are obtained from “service description” files that must be constructed by the user in advance. Another important difference between GREEDY and the DARQ optimizer is that GREEDY is adaptive, whereas DARQ follows the more traditional approach of static query optimization. DARQ implements an additional type of optimization in the form of query rewriting rules, which perform transformations such as the merging of basic graph patterns and the pushing of FILTER expressions into subqueries; this type of optimization is currently not implemented in SHARE. For Sesame, [165] describes a join procedure that makes use of semi-joins and row blocking. Query optimization is a subject of future work for the SemWIQ system.

The GREEDY algorithm differs from most query optimizers because the processes for planning and executing the query are interleaved. In the literature, this type of approach is termed adaptive query execution. Two prior examples of adaptive query systems are Tukwila [166] and Telegraph [167], both of which are relational systems. Tukwila uses traditional System R-style optimization, but additionally allows for the creation of partial plans when adequate statistics are not available, and for replanning when cardinality estimations prove to be inaccurate. Telegraph is a more radical departure from traditional query processing. It replaces the standard tree-of-operators representation for a query plan with a circular structure called an eddy that connects all of the operators. The idea of the eddy is to dynamically change the routing of the

---

8Strictly speaking, this is only true for SADI services; there is no mechanism for batching queries to SPARQL endpoints.
3.3 Key Points of SHARE Query Resolution

3.3.1 Resolving Triple Patterns

In Section 2.4, we described the procedure that SHARE uses to answer a SPARQL query over a distributed set of services; we summarize the key points of the procedure here. SHARE answers a query by resolving each triple pattern of the query in sequence. A triple pattern \( (s, p, o) \) may be resolved in either the forward direction or the reverse direction:

- If a triple pattern is resolved in the **forward direction**, all services that generate \( p \) are invoked using the bindings of \( s \) as the input URI(s), and the resulting output data is loaded into the local triple store. A service generates \( p \) if it outputs triples of the form \( \langle \text{input URI}, p, * \rangle \), where * represents any URI or literal value. If \( o \) is an unbound variable, the set of distinct values returned for * become the bindings of \( o \).

- If a triple pattern is resolved in the **reverse direction**, all services that generate \( \text{inverse}(p) \) are invoked using the bindings of \( o \) as the input URI(s), and the resulting output data is loaded into the local triple store. \( \text{inverse}(p) \) represents an inverse property of \( p \), meaning that statements of the form \( (X, p, Y) \) and \( (Y, \text{inverse}(p), X) \) are logically equivalent. A service generates \( \text{inverse}(p) \) if it outputs triples of the form \( \langle \text{input URI}, \text{inverse}(p), * \rangle \). If \( s \) is an unbound variable, the set of distinct values returned for * become the bindings of \( s \).

Once all of the triple patterns have been resolved, the local triple store will contain a sufficient set of RDF triples to produce the solutions for the query. As the final step, the user’s original query is issued against the local triple store and the solutions are displayed.

3.3.2 OWL Inferences About Predicates

SHARE implicitly makes use of OWL reasoning during query resolution. Whenever a new predicate or OWL class is encountered within a triple pattern, the query engine downloads the OWL ontology file for that predicate/class and loads it into the local reasoner. One advantage of using OWL reasoning during query processing is that the mapping from predicates to services is more intelligent. The system is aware of which predicates are synonyms, inverses, or connected by a parent/child relationships (i.e. subproperties), and can perform the necessary translations between related predicates implicitly. For example, if two predicates that describe the relationship between a drug and a target (e.g. “hasTarget” and “hasDrugTarget”) are equivalent, the user may use either predicate within a query and be certain that all of the relevant services will be discovered and invoked.

However, this additional intelligence comes at a cost. First, downloading ontology files during query resolution entails additional data transfer, and thus additional time when resolving triple patterns. Second, the computation of inferences can be expensive, depending on the number of ontology rules (e.g. synonym relationships between predicates) and the amount of data in the local triple store. For the evaluation of the optimizer in Section 3.7, the author has disabled OWL inferencing in SHARE. This allows the data retrieval aspect of the optimization problem to be studied in isolation.

---

9. By convention, the inverse properties of \( p \) (if any exist) are defined in the OWL ontology that defines \( p \).
10. By convention, the OWL ontology for a given predicate/class may be downloaded by resolving its URI.
3.4 The Distributed SPARQL Query Optimization Problem

3.4.1 An Illustrative Example

The ordering of triple patterns in a query does not affect the logical meaning of the query. For example, Fig. 3.1a and Fig. 3.2a show two equivalent SPARQL queries with different orderings; both versions ask the system to “List all motifs that occur in Poecilia reticulata (guppy fish) proteins” [168]. While the ordering of triple patterns does not change the meaning, it does change the steps that SHARE uses to gather the required data, and thus it can have a significant effect on query efficiency. To illustrate, Fig. 3.1b and Fig. 3.2b depict the alternate steps used by SHARE to resolve the two alternate orderings of the guppy query. The first ordering (Fig. 3.1b) is a bad strategy for resolving the query because there is a “fan-out” effect in Steps 1 and 2; there are a large number of PROSITE motifs (1,456), and each of those motifs is found in large number of proteins. As a result, the query engine sends 166,986 protein URIs to the services that map proteins to organisms in Step 3, and the query continues running for more than an hour. (After an hour, the query was manually aborted). In contrast, the second ordering of the query (Fig. 3.2b) begins by retrieving the 48 proteins that belong to Poecilia reticulata. Only a small amount of data is gathered (10,480 triples) to answer the query, and the query completes successfully in about 30 seconds.

<table>
<thead>
<tr>
<th>Query Step</th>
<th>Description</th>
<th>Triple Pattern</th>
<th>Time (ms)</th>
<th>Variable Bindings Assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Retrieve a list of all PROSITE motifs</td>
<td>(?motif, rdf:type, prosite:Site)</td>
<td>2,749</td>
<td>?motif ⇒ 1,456 bindings</td>
</tr>
<tr>
<td>2</td>
<td>For each motif from Step 1, retrieve all proteins that contain that motif</td>
<td>(?motif, bio2rdf:xUniProt, ?protein)</td>
<td>170,639</td>
<td>?protein ⇒ 166,986 bindings</td>
</tr>
<tr>
<td>3</td>
<td>For each protein from Step 2, retrieve the organism it belongs to</td>
<td>(?protein, core:organism, taxonomy:8081)</td>
<td>&gt; 1 hour (query aborted)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 3.1: Description of execution steps for an inefficient query ordering, as depicted in Fig. 3.1b.
3.4. The Distributed SPARQL Query Optimization Problem

PREFIX bio2rdf: <http://bio2rdf.org/ns/bio2rdf#>
PREFIX taxonomy: <http://bio2rdf.org/taxonomy:>
PREFIX core: <http://purl.uniprot.org/core/>
PREFIX prosite: <http://bio2rdf.org/ns/prosite#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?motif
WHERE {
  ?motif rdf:type prosite:Site .
}

(a) SPARQL query

(b) execution plan as graph traversal

Figure 3.1: Example of an inefficient ordering of triple patterns for a query, along with its associated execution plan. A textual description of each step in the execution plan is provided in Table 3.1. The query asks the system to “List all motifs that occur in Poecilia reticulata (guppy fish) proteins.”
3.4. The Distributed SPARQL Query Optimization Problem

PREFIX bio2rdf: <http://bio2rdf.org/ns/bio2rdf#>
PREFIX taxonomy: <http://bio2rdf.org/taxonomy:>
PREFIX core: <http://purl.uniprot.org/core/>
PREFIX prosite: <http://bio2rdf.org/ns/prosite#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?motif
WHERE {
  ?motif rdf:type prosite:Site .
}

(a) SPARQL query

(b) execution plan as graph traversal

Figure 3.2: Example of an efficient input ordering for a query, along with its associated execution plan. A textual description of each step in the execution plan is provided in Table 3.2. The query asks the system to “List all motifs that occur in Poecilia reticulata (guppy fish) proteins.”
3.4. The Distributed SPARQL Query Optimization Problem

<table>
<thead>
<tr>
<th>Query Step</th>
<th>Description</th>
<th>Triple Pattern</th>
<th>Time (ms)</th>
<th>Variable Bindings Assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Retrieve all proteins that belong to Poecilia reticulata</td>
<td>(?protein, core:organism, taxonomy:8081)</td>
<td>615</td>
<td>?protein (\Rightarrow) 48 bindings</td>
</tr>
<tr>
<td>2</td>
<td>Retrieve the motifs contained by each protein from Step 1</td>
<td>(?motif, bio2rdf:xUniProt, ?protein)</td>
<td>9,112</td>
<td>?motif (\Rightarrow) 8 bindings</td>
</tr>
<tr>
<td>3</td>
<td>For each motif from Step 2, retrieve its type to determine if it is a PROSITE motif(^1)</td>
<td>(?motif, rdf:type, prosite:Site)</td>
<td>19,099</td>
<td>N/A</td>
</tr>
</tbody>
</table>

\(^1\) This step is necessary because the bio2rdf:xUniProt predicate is also used to connect genes to UniProt proteins.

Table 3.2: Description of query execution steps for an efficient query ordering, as depicted in Fig. 3.2b.

3.4.2 Assumptions

The goal of the optimization algorithm is to determine the best ordering for the triple patterns in a distributed SPARQL query; that is, the ordering which will answer the query in the shortest amount of time. To simplify the problem, we assume that the execution time of a query is determined solely by:

1. The time spent by services processing requests
2. The time spent transferring data (service requests and responses) over the network

The use of OWL inferencing in SHARE can add a significant amount of local processing time to query resolution. However, as the optimization of OWL reasoning is a challenging problem on its own, for the purposes of this chapter we disable OWL reasoning in SHARE and assume that the cost of local query processing is zero. In addition, we assume that the following constraints hold for resolving triple patterns:

1. Variables may not appear in the predicate positions of triple patterns.
2. Triple patterns may only be resolved when either the subject or the object is bound.

These constraints are not unique to SHARE; they are also applicable to the earlier DARQ \[^{152}\] system, which uses the same predicate-based method for mapping triple patterns to services. Under constraints 1 and 2 above, the resolution of a distributed SPARQL query may be visualized as a graph traversal, as depicted in Fig. 3.1b. Each constant in the query acts an independent starting point for one branch of the traversal, and the traversal is complete when all edges have been visited. The traversal of an edge is in effect a distributed join operation, where the outer table is a list of URIs (variable bindings), and the inner table is the set of triples that are “stored”\(^{11}\) by the relevant services. Thus, the optimization problem discussed in this paper is a distributed version of the well-known join-order problem \[^{159},160\].

\(^{11}\) The word “stored” is quoted here because triples may be generated computationally by the relevant services. Conceptually, the set of triples “stored” by a computational service is the full set of triples generated by invoking the service with all possible input URIs.
3.5 Secondary Optimization: Intersections of Variable Bindings

This section describes an optimization for the assignment of variable bindings that is independent of the query evaluation algorithm (GREEDY) presented in Section 3.6. The optimization is implemented by a small change to the processPattern method from Chapter 2 (listing 2.2), as shown in Fig. 3.3. In the original method, the bindings for a variable are determined when the first pattern that references the variable is resolved; thereafter, the bindings of the variable are fixed. In the optimized method, the bindings that are determined for a variable are intersected with any existing bindings. To demonstrate the difference, consider answering the query shown in Fig. 3.4, which asks “What proteins in the post-synaptic membrane are targeted by the drug Benzthiazide?”, using the original processPattern method. Here the variable of interest is ?protein, which occurs in three different patterns (patterns 1, 3, and 4). When pattern 1 is resolved, ?protein is assigned 965 proteins that are located in the post synaptic membrane. When pattern 2 is resolved, ?target is assigned 5 drug target records for Benzthiazide. The critical step occurs in pattern 3. When pattern 3 is resolved, the 5 bindings for ?target (drug target records) are mapped to corresponding UniProt IDs. However, because ?protein has already been assigned a set of bindings in pattern 1, the 965 existing bindings for ?protein are left unmodified. As a result, resolving pattern 4 requires retrieving the names (rdfs:label) of 965 proteins, which requires ~ 2 minutes. The query time can be significantly reduced if the system computes the intersection of the bindings for ?protein from patterns 1 and 3. Using this approach, ?protein is reduced to a single binding and thus pattern 4 is resolved with 1 input. Fig. 3.5a and Fig. 3.5b depict the execution of the query without and with the variable bindings optimization, respectively.

We assume here that pattern 3 is resolved in the forward direction. The reverse direction is also possible, but it does not make any difference for the purposes of this discussion.
3.5. Secondary Optimization: Intersections of Variable Bindings

```plaintext
processPattern(s, p, o, bindings)

subjects = bindings[s]
objects = bindings[o]

if (subjects == null && objects == null)
    warn "cannot resolve pattern where both subject and object are unbound variables"
else if (objects == null)
    invokeServicesByProperty(subjects, p)
else if (subjects == null)
    invokeServicesByProperty(objects, inverse(p))
else
    // both subject and object are already bound
    invokeServicesByProperty(subjects, p)

for each triplet t in localTripleStore.getMatchingTriples(bindings[s], bindings[p], bindings[o])
    if(subjects == null)
        bindings[s].add(t.subject)
    if(objects == null)
        bindings[o].add(t.object)
```

(a) original processPattern method

```plaintext
processPatternOptimized(s, p, o, bindings)

subjects = bindings[s]
objects = bindings[o]

if (subjects == null && objects == null)
    warn "cannot resolve pattern where both subject and object are unbound variables"
else if (objects == null)
    invokeServicesByProperty(subjects, p)
else if (subjects == null)
    invokeServicesByProperty(objects, inverse(p))
else
    // both subject and object are already bound
    invokeServicesByProperty(subjects, p)

newBindings = empty hash table

for each triplet t in localTripleStore.getMatchingTriples(bindings[s], bindings[p], bindings[o])
    newBindings[s].add(t.subject)
    newBindings[o].add(t.object)

if(subjects == null)
    bindings[s] = newBindings[s]
else
    bindings[s] = intersection(bindings[s], newBindings[s])

if(objects == null)
    bindings[o] = newBindings[o]
else
    bindings[o] = intersection(bindings[o], newBindings[o])
```

(b) optimized processPattern method

Figure 3.3: A modification to the processPattern method from Chapter 2 (listing 2.2) which implements the variable bindings optimization described in Section 3.5. Part a) shows the original pseudocode and part b) shows the modified pseudocode.
3.5. Secondary Optimization: Intersections of Variable Bindings

PREFIX drug: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugs/>
PREFIX drugbank: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX core: <http://purl.uniprot.org/core/>
PREFIX go: <http://bio2rdf.org/go:>

SELECT ?proteinName
WHERE {
  ?target drugbank:swissprotId ?protein.
}

(a) SPARQL query

(b) SPARQL query, depicted graphically

Figure 3.4: An example query for SHARE, used to illustrate variable bindings optimization described in Section 3.5. The query asks “What proteins in the post-synaptic membrane are targeted by the drug Benzthiazide?” go:0045211 is the Gene Ontology term for “post synaptic membrane”, and drug:DB00562 is the identifier for Benzthiazide.
3.5. Secondary Optimization: Intersections of Variable Bindings

Figure 3.5: The execution plans used by SHARE to solve the query from Fig. 3.4 without and with the variable bindings optimization described in Section 3.5. Textual descriptions of the two execution plans are given in Table 3.3 and Table 3.4, respectively.
### 3.5. Secondary Optimization: Intersections of Variable Bindings

<table>
<thead>
<tr>
<th>Query Step</th>
<th>Description</th>
<th>Triple Pattern</th>
<th>Time (ms)</th>
<th>Variable Bindings Assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Retrieve all proteins that have been annotated with Gene Ontology term go:0045211 (post synaptic membrane)</td>
<td>(?protein, core:classifiedWith, go:0045211)</td>
<td>2,949</td>
<td>?protein ⇒ 965 bindings</td>
</tr>
<tr>
<td>2</td>
<td>Retrieve all targets of the drug DB00562 (Benzthiazide)</td>
<td>(drug:DB00562, drug-bank:target, ?target)</td>
<td>3,896</td>
<td>?target ⇒ 5 bindings</td>
</tr>
<tr>
<td>3</td>
<td>Retrieve the equivalent UniProt identifier for each target from Step 2</td>
<td>(?target, drug-bank:swissprotId, ?protein)</td>
<td>2,936</td>
<td>No bindings are assigned.</td>
</tr>
<tr>
<td>4</td>
<td>For each protein from Step 1, retrieve its name.</td>
<td>(?protein, rdfs:label, ?proteinName)</td>
<td>128,363</td>
<td>?name ⇒ 965 bindings</td>
</tr>
</tbody>
</table>

1 No bindings are assigned to ?protein in this step, because ?protein has already been assigned a set of bindings in Step 1.

Table 3.3: Description of execution steps without the variable bindings optimization, as depicted in Fig. 3.5a.

<table>
<thead>
<tr>
<th>Query Step</th>
<th>Description</th>
<th>Triple Pattern</th>
<th>Time (ms)</th>
<th>Assigned Variable Bindings</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Retrieve all proteins that have been annotated with Gene Ontology term go:0045211 (post synaptic membrane)</td>
<td>(?protein, core:classifiedWith, go:0045211)</td>
<td>2,655</td>
<td>?protein ⇒ 965 bindings</td>
</tr>
<tr>
<td>2</td>
<td>Retrieve all targets of the drug DB00562 (Benzthiazide)</td>
<td>(drug:DB00562, drug-bank:target, ?target)</td>
<td>3,673</td>
<td>?target ⇒ 5 bindings</td>
</tr>
<tr>
<td>3</td>
<td>Retrieve the equivalent UniProt identifier for each target from Step 2</td>
<td>(?target, drug-bank:swissprotId, ?protein)</td>
<td>2,936</td>
<td>?protein ⇒ 1 binding</td>
</tr>
<tr>
<td>4</td>
<td>For each protein from Step 3 retrieve the protein’s name.</td>
<td>(?protein, rdfs:label, ?proteinName)</td>
<td>12,020</td>
<td>?name ⇒ 1 binding</td>
</tr>
</tbody>
</table>

1 This step is where the optimization takes place. When the pattern is resolved, 5 distinct values are obtained for ?protein. This set of 5 solutions is then intersected with the existing 965 bindings for ?protein from Step 1. The two sets have one value in common, and thus ?protein is assigned 1 binding.

2 For each protein that is both a target of Benzthiazide and located in post synaptic membrane.

Table 3.4: Description of execution steps with the variable bindings optimization, as depicted in Fig. 3.5b.
3.6 The GREEDY Optimization Algorithm

3.6.1 Overview
In this section, we will describe an algorithm for evaluating distributed SPARQL queries called GREEDY. In the default method of query evaluation for SHARE, which we will hereafter call BASIC, triple patterns are resolved in the same order that they occur in the query. In addition, BASIC resolves all triple patterns in the forward (left-to-right) direction by default, unless only the reverse direction is possible (i.e. the subject is an unbound variable). The pseudocode for the default query evaluation procedure is shown in listing 2.2. GREEDY uses a more sophisticated approach for determining the ordering and directions of the triple patterns. At each step, the costs for all unvisited triple patterns are estimated and compared pairwise, and the pattern that is cheapest overall is selected for resolution. If a particular pattern is resolvable in both the forward and reverse directions (i.e. both the subject and object of the pattern are bound), then the cost for both directions is estimated and the overall cost of pattern is considered to be the lesser of the two. The pseudocode for the main procedure of GREEDY is shown in listing 3.1.

While the main procedure of GREEDY is straightforward, the operation of compareCost(pattern1, pattern2) remains to be explained. While no statistics about the services are directly available, the algorithm must somehow estimate the relative costs of the triple patterns. In GREEDY, the relative cost of a triple pattern is estimated in one of two ways:

1. Using statistics about predicates that have been learned from previous queries
2. Using the number of bindings for the subject/object of the triple pattern

The latter method acts as a fallback in the case where no statistics have yet been learned from previous queries. In the next section, we describe details for estimating the cost of a triple pattern.

3.6.2 Cost Estimation for Query Patterns
Cost Equation
GREEDY estimates the cost of retrieving data for a triple pattern by the following equation:

\[
\text{estimatedTime} = \text{baseTime}(p, \text{dir}) + \text{numInputs} \times \text{timePerInput}(p, \text{dir})
\]  

(3.1)

where

- \text{estimatedTime} is the expected time to retrieve the data
3.6. The GREEDY Optimization Algorithm

- **baseTime**\( (p, \text{dir}) \) is the sum of the round-trip network latencies\(^{13}\) for all services that resolve predicate \( p \) in the direction \( \text{dir} \)

- **numInputs** is the number of input URIs that are sent to each service

- **timePerInput**\( (p, \text{dir}) \) is the time required to receive the response data for a single input URI, from all services that resolve predicate \( p \) in the direction \( \text{dir} \)

The resolution of a triple pattern \((s, p, o)\) is in effect a join operation between the local triple store and the triples stored by services that generate \( p \). However, in contrast to traditional cost estimations for joins, which are based on the sizes of the input tables, Eq. (3.1) is based on service response times. There are two reasons for this. First, the time required to transfer data over the network is a significant cost that is not present in a single database scenario, and it is a cost that may vary for each data source. Second, the data sources can be computational services. The amount of processing time required by different algorithms is likely to vary greatly (e.g. a BLAST query vs. a protein structure prediction), and will also vary based on the CPU power and load of the servers that are used to perform the analyses. These factors are represented in Eq. (3.1) by the timePerInput\( (p, \text{dir}) \) factor.

Learning Statistics from Queries

After each triple pattern is resolved, GREEDY records a sample containing the following information:

- predicate URI
- direction of resolution (forward or reverse)
- number of inputs (i.e. number of bindings of the subject/object of the triple pattern)
- total response time (in milliseconds)

The total response time is the time required to obtain all data for a triple pattern from the available services. The information listed above constitutes a single sample in the statistics database. At periodic intervals, the available samples are used to compute the values for baseTime\( (p, \text{dir}) \) and timePerInput\( (p, \text{dir}) \) using linear regression, as depicted in Fig. 3.6. Once the baseTime and timePerInput values have been computed, GREEDY is able to use them to estimate the times for resolving triple patterns, as per Eq. (3.1).

Special Cases for Cost Estimation

In order to compute a regression line for a given predicate and direction, at least two samples with distinct \( x \) values (i.e. numInputs values) must be recorded in the statistics database. There are two cases where summary statistics will not be available for a given predicate/direction combination (denoted “\((p, \text{dir})\)” below):

1. **No samples have been recorded for \((p, \text{dir})\):** If available, values for averageBaseTime and averageTimePerInput will be used instead. averageBaseTime and averageTimePerInput are calculated over the baseTime and timePerInput values for all predicates in the statistics database:

   \[
   \text{averageBaseTime} = \frac{\sum \text{baseTime}(p, \text{dir})}{N}
   \]

\(^{13}\)baseTime\( (p, \text{dir}) \) also includes the cost of establishing a TCP connection.
Figure 3.6: An example regression line, computed to determine the cost parameters baseTime(bio2rdf:xUniProt, reverse) and timePerInput(bio2rdf:xUniProt, reverse), as described in Section 3.6.2. The Bio2RDF project uses the bio2rdf:xUniProt predicate to denote a cross reference from an entity (e.g. a gene) to a UniProt protein record. The prefix “bio2rdf:” is an abbreviation for [http://bio2rdf.org/ns/bio2rdf#](http://bio2rdf.org/ns/bio2rdf#).
3.7. Evaluation

\[
\text{averageTimePerInput} = \frac{\sum \text{timePerInput}(p, \text{dir})}{N}
\]

where \( N \) is the number of predicate/direction combinations that have computed values for baseTime and timePerInput.

These values are intended to represent the characteristics of a predicate with average performance. averageBaseTime and averageTimePerInput will only be available if a regression line has been computed for at least one predicate / direction combination in the statistics database. If this is not the case, GREEDY will use numInputs (i.e. the number of subject/object bindings) as the cost of the triple pattern. This is equivalent to the assumption that all predicate / direction combinations perform equivalently when given the same number of inputs.

2. **One or more samples with the same x value (numInputs) have been recorded for \((p, \text{dir})\):** Let the common value of numInputs across the samples be \( X \). An average response time will be computed for the samples, and this value will be used to estimate the cost of any future triple patterns where numInputs is exactly equal to \( X \). Otherwise, the cost of the triple pattern will be computed from averageBaseTime/averageTimePerInput or numInputs, as per Case 1.

3.7 Evaluation

3.7.1 Evaluation Procedure

At the current time, no benchmarks are available for distributed SPARQL systems. This is not surprising, given that benchmarking of distributed systems is considerably more difficult than for monolithic database systems. Whereas the performance of monolithic systems can be measured independently with standardized data sets, distributed systems must be compared as a group, in order to control for factors such as changing data sources, server load, connection bandwidth, and volume of traffic on the network. Here we evaluate the performance of GREEDY in a simple before-and-after manner; that is, we measure the performance differences between BASIC (optimizer off) and GREEDY (optimizer on) with increasing amounts of training, without comparison to other systems.

In order to base the evaluation of GREEDY on realistic use cases, the author translated 6 bioinformatics queries found in the literature into equivalent SHARE queries (Fig. 3.7). The original queries were taken from the publications of other bioinformatics integration projects, namely TAMBIS [168], GALAXY [169], BioMART [119], and the Linked Open Drug Data (LODD) [91] project. These systems were the most suitable source for third-party queries because they target many of the same data sources as SHARE. Nevertheless, some translations could not be made perfectly. For instance, Query 3 (Fig. 3.7) originally used the UCSC genome database to obtain exon boundaries and SNP coordinates, while in the SHARE version, the SNP annotations provided by UniProt are used instead. Notes are provided in Fig. 3.7, indicating where such differences exist.

To evaluate the learning aspect of the GREEDY algorithm, the queries of Fig. 3.7 were divided into 3 training queries and 3 test queries. The assignment of queries into the two sets was based on the overlap in subject matter between the queries. Queries 1 & 2 involve SNPs, and Queries 3 & 4 involve drugs and associated targets, Queries 5 & 6 involve biological pathways. For query pairs 1/2 and 5/6, one query was randomly chosen as the training query and the other became the testing query. For query pair 3/4, Query 3 was explicitly chosen as the training query because it contains only one constant; this implies that Query 3 has only one possible execution plan, and thus optimization would have no effect. More will be said about queries with a single constant in the conclusion (Section 3.9).
3.7. Evaluation

For each training query, 3 additional training variants were generated by substituting the constants in the query with randomly selected replacement values. For example, the training variants for Query 1 were generated by changing the target protein. Similarly variants for queries 3 and 5 were generated by changing the target drug and biological pathway, respectively. Generating variants of the training queries was necessary in order to provide adequate training data for GREEDY. If the optimizer were trained by running the same queries repeatedly, the predicates in the training queries would always be resolved using the same number of subject/object bindings (i.e. service inputs), and thus no regression lines could be computed. The variants used for each training query are listed in Appendix B.1. Each training query was run with GREEDY rather than BASIC. This most accurately reflects the real usage of SHARE, where the optimizer is enabled for all queries.

PREFIX uniprot: <http://bio2rdf.org/uniprot:>
PREFIX core: <http://purl.uniprot.org/core/>
PREFIX core2: <http://bio2rdf.org/core:>
PREFIX rdf: <http://www.w3.org/1999/02/rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?substitution ?start ?end ?dbCrossRef
WHERE {
  uniprot:P01344 core:annotation ?annotation . # uniprot:P01344 = "IGF–II gene"
  ?annotation rdf:type core2:Natural_Variant_Annotation .
  ?annotation rdfs:seeAlso ?dbCrossRef .
  ?annotation core:substitution ?substitution .
  ?range core:begin ?start .
  ?range core:end ?end .
}

(a) Query 1 (training query): “Find non-synonymous single nucleotide polymorphisms (SNPs) within coding exons of the human insulin-like growth factor II (IGF-II) gene”. The original query from the GALAXY project [169] did not specify that the SNPs should be non-synonymous; this was a limitation of using UniProt to obtain SNP information, rather than the UCSC genome database as per the original query.

(b) Query 1 as graph

![Diagram of Query 1 as graph](attachment:image.png)
3.7. Evaluation

WHERE {
    # gene IDs
    probeset:53701_at affy:xEnsembl ?geneID .
    # protein IDs
    probeset:53701_at affy:xSwissProt ?proteinID .
    # GO terms
    # SNPs
    ?annotation rdf:type core2:Natural_Variant_Annotation .
    ?annotation rdfs:seeAlso ?snpID .
})

(c) Query 2 (test query): “Map a microarray probeset to associated information: ENSEMBL gene IDs, SwissProt protein IDs, SNPs, and GO terms.”, a use case from the BioMART project [119]. As in Query 1, the results are limited to non-synonymous SNPs.
3.7. Evaluation

PREFIX drug: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugs/>  
PREFIX drugbank: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/>  
PREFIX diseasome: <http://www4.wiwiss.fu-berlin.de/diseasome/resource/diseasome/>  
PREFIX bio2rdfs: <http://bio2rdfs.org/ns/bio2rdfs/>  
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?targetProtein ?gene ?disease ?diseaseName  
WHERE {  
}

(e) **Query 3 (training query):** “For each target protein of the drug Spironolactone, show the gene that encodes the protein, and any known disease associations for the gene”, from the Linked Open Drug Data (LODD) project [91]. The original “query” that appears in [170] is answered interactively, by browsing across the LODD datasets from within a web browser. The SHARE query answers the same question using a distributed query across the DrugBank, HGNC, and Diseasome SPARQL endpoints. The subject drug of the original query has been changed from Varenicline to Spironolactone to avoid overlap of training and testing queries during the optimizer evaluation.
3.7. Evaluation

PREFIX drug: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugs/>
PREFIX drugbank: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX core: <http://purl.uniprot.org/core/>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>

SELECT ?targetProtein ?alzProtein ?alzProteinName
WHERE {
  drug:DB01273 drugbank:target ?target . # drug:DB01273 = "Varenicline"
  ?target drugbank:swissprotId ?targetProtein .

  ?alzProtein rdfs:seeAlso omim:104300 . # omim:104300 = "Alzheimer’s Disease"
  ?alzProtein dc:title ?alzProteinName .

  ?participant owl:sameAs ?alzProtein .
}

(g) Query 4 (test query): “Are there any protein-protein interactions between Alzheimer-associated proteins and the targets of Varenicline?”, from the Linked Open Drug Data project. The original “query” posed in [170] was answered by browsing hypotheses in the AlzSWAN KnowledgeBase [171]. In the SHARE version, the question is answered by a distributed query across the UniProt and DrugBank SPARQL endpoints.
3.7. Evaluation

Figure 3.7: SHARE translations of six queries from other bioinformatics integration projects, including TAMBIS, GALAXY, BioMART, and Linked Open Drug Data. These queries are used for the evaluation of the GREEDY optimization algorithm.
The overall idea of the evaluation is to measure the performance difference between BASIC (optimizer off) and GREEDY (optimizer) with increasing amounts of training. However, the ordering of triple patterns in an input query can have a large effect on what this performance difference actually is. For example, if a “smart” user composes a query that already has the optimal ordering of triple patterns, then there will be no difference at all between BASIC and GREEDY for the performance of that query. For the optimizer evaluation described here, we will make the simplifying assumption that each input query ordering is equally likely. Ideally, we would like to compare the performance of BASIC and GREEDY for all possible orderings of each test query. This would provide us with the complete range of execution times that are possible for a given query; that is, how slow the query can be and how fast it can be, and how the possible orderings for the query are distributed between these two extremes. Unfortunately, the exhaustive approach is not practical, even for the relatively simple queries presented in Fig. 3.7; for example, Query 2 has 8 triple patterns, and thus has 8! = 40,320 possible orderings. Instead, execution times were measured for 5 randomly generated test query orderings, with the understanding that these orderings are merely a sample from the population of possible query orderings.

The pseudocode for the benchmarking procedure is shown in listing 3.2. Three trials were recorded for each test query, in order to give an indication of the amount of variance that occurs between executions of the same query due to varying server load and data transfer times. The results in the following section demonstrate that this variance is small relative to the changes in query times that result from reordering triple patterns in the query.

### 3.7.2 Results

#### Benchmarking Results

Figure 3.8 shows the execution times for the 5 randomly generated orderings of Test Queries 2, 4, and 6. (For a complete list of generated orderings for each query, see Appendix B.2.) Each group of bars in Fig. 3.8 represents a set of execution times for one particular ordering of a query. Within each group, the execution times were generally expected to decrease from left-to-right, beginning with the execution time for BASIC (optimizer off) and proceeding through execution times for GREEDY (optimizer on) with increasing numbers of training runs. A timeout of an hour was imposed on each query to ensure that the overall experiment would complete in a reasonable amount of time.

The solution sets generated for all trials of Queries 2, 4, and 6 were identical, regardless of the ordering of triple patterns, and regardless of whether the queries were run under BASIC or GREEDY with 0-4 training runs. This was verified by recording the solution set for each query execution to a separate file, and comparing the files with the unix ‘fdupes’ utility for identifying

```plaintext
FOR trainingRun = 0 to 5
    IF (trainingRun > 0)
        FOR trainingQuery = { Query 1, Query 3, Query 5 }
            Run variant trainingRun of trainingQuery and record stats
    FOR testQuery = { Query 2, Query 4, Query 6 }
        FOR testQueryOrdering = 0 to 9
            IF (trainingRun == 0)
                FOR trial = 0 to 2
                    Run testQuery with BASIC and time
                FOR trial = 0 to 2
                    Run testQuery with GREEDY and time
Listing 3.2: Pseudocode for the benchmarking procedure for GREEDY
```
3.7. Evaluation

(a) Execution times for Query 2

(b) Execution times for Query 4
3.7. Evaluation

Figure 3.8: Results of GREEDY evaluation: a comparison of query execution times for Queries 2, 4, and 6 of Fig. 3.7. Each query is executed using 5 randomly generated orderings, and for each ordering, query times are recorded for BASIC (optimizer off) and GREEDY (optimizer on) with varying numbers of training runs.
3.7. Evaluation

duplicate files. The lines of each solution file were sorted alphabetically prior to comparison, as some trials would generate the rows of the solutions table in different orders.

In order to fully understand the decisions made by the trained versions of GREEDY in the discussion below, it is necessary to see the state of the predicate statistics after each training run. These states are depicted in Fig. 3.9 using two types of graphs:

1. scatterplots (left): show average response times for predicates which only have samples for a single x (numInputs) value.

2. bar graphs (right): show the slope (timePerInput) and intercept (baseTime) for predicates that have sufficient samples for a regression line.

The reader will observe from the bar graphs of Fig. 3.9 that for most of the predicates, a regression line is only ever computed in one of the two possible directions. This happens because the structure of the training queries (Fig. 3.7) only permits resolving these predicates in one direction. For example, in Query 3, the drugbank:hgncId predicate is always resolved in the forward direction because the path of traversal must begin with the sole constant in the query, drug:DB00421. The sadl:isCausallyRelatedWith is a special case; although it is always resolved in the forward direction during training, it is its own inverse property, and so the same regression line describes both directions.

Results for Query 2 (Fig. 3.8a)

Over all trials, we observe 3 distinct query times: ∼45 seconds (best), ∼100 seconds (fair), and timeout after 60 minutes (worst). Examination of the query logs shows that it is the treatment of one particular triple pattern that determines the outcome for each trial:

?annotation rdf:type core2:Natural_Variant_Annotation

One of the conventional uses of rdf:type is to indicate database record types. In this particular triple pattern, Natural_Variant_Annotation represents the type of record that is used to store amino acid sequence variations in the UniProt database\textsuperscript{14}. When the pattern is resolved in the reverse direction, a service call is made which retrieves all Natural_Variant_Annotation records from Bio2RDF’s UniProt SPARQL endpoint. At the current time, there are 70,629 such records, and retrieving a list of their identifiers requires about 55 seconds. While resolving the rdf:type pattern in the reverse direction significantly increases the query time, it is not by itself disastrous. More serious performance problems (i.e. timeouts) occur when the query engine proceeds to use the 70,629 bindings for ?annotation as input for solving a later query pattern. To illustrate, Fig. 3.10a shows the sequence of steps used by BASIC for Query Ordering 3 which lead to a query timeout.

The GREEDY heuristic avoids the use of large input sets when resolving query patterns, and thus turning on the optimizer for Query Orderings 3 and 4 produces an immediate and significant improvement. However, without training, the choices of GREEDY are still not optimal. The GREEDY heuristic predicts that resolving the rdf:type pattern in the reverse direction will be an fast operation, since it requires sending only one input (Natural_Variant_Annotation) to the relevant services. The implicit assumption of the GREEDY heuristic is that all predicates will require the same amount of time to resolve, given the same number of inputs. This assumption does not hold true for the rdf:type predicate, and so GREEDY requires training in order to make the correct decision. Figure Fig. 3.10b and Fig. 3.10c show the steps followed by GREEDY for Query Ordering 3 which lead to a query timeout.

\textsuperscript{14}The majority of these variations are single amino acid polymorphisms (SAPs), which are in turn due to single nucleotide polymorphisms (SNPs) in the coding DNA. In the UniProt database, the category of naturally occurring mutations also includes disease associated mutations and RNA editing events.
3.7. Evaluation

(a) 1 Training Run: Average Predicate Response Times

(b) 2 Training Runs: Average Predicate Response Times

(c) 2 Training Runs: Regression Line Parameters

(d) 3 Training Runs: Average Predicate Response Times

(e) 3 Training Runs: Regression Line Parameters
3.7. Evaluation

Figure 3.9: State of predicate statistics during the experiment. **Left:** Scatter plots showing the average response times of various predicates for a fixed number of inputs. The predicate/direction pairs included these graphs have the same x value (i.e. numInputs) for all samples, and thus no regression line is computable. **Right:** Bar graphs depicting the slope (timePerInput) and intercept (baseTime) for all predicates for which a regression line was computable. Note that there is no bar graph for 1 training run because a regression line can only be computed from two or more samples; each predicate is used in only one training query, and so obtaining adequate samples entails two or more training runs.

(f) 4 Training Runs: Average Predicate Response  (g) 4 Training Runs: Regression Line Parameters Times
the forward direction. Finally, the reader will notice from Fig. 3.8a (and also Fig. 3.8c) that GREEDY always requires two training runs before there is any improvement in query times. The reason for this is that a minimum of two samples are required to compute a regression line. Until a regression line has been computed for at least one predicate, all patterns are compared by the number of input bindings.

The results for each random query ordering (i.e. group of bars) in Fig. 3.8a differ only with respect to the query times under BASIC. In query orderings 0 and 2, the patterns were ordered such that BASIC resolved the rdf:type in the reverse direction, but did not use the large number of resulting bindings for ?annotation to resolve a later pattern. In query ordering 1, the patterns were ordered such that the rdf:type pattern was resolved in the forward direction. In query orderings 3 and 4, the patterns were ordered such that the rdf:type pattern was resolved in the reverse direction and the resulting bindings of ?annotation were used as input for a later pattern. (The complete list of randomly generated query orderings for the experiment is provided in Appendix B.2.) The results for query ordering 1 illustrate an important caveat of GREEDY, which is that query times prior to training may actually be worse than for BASIC, in the case that a “smart” user specifies the query patterns in an optimal order or is simply lucky.
3.7. Evaluation

(a) The steps followed by BASIC, in an attempt to answer Test Ordering 3 of Query 2 (Appendix B.2.1). The query times out after 60 minutes.

(b) The steps followed by GREEDY with 0 training runs, in order to answer Test Ordering 3 of Query 2 (Appendix B.2.1).

(c) The steps followed by GREEDY with 2 training runs, in order to answer Test Ordering 3 of Query 2 (Appendix B.2.1).

Figure 3.10: Execution plans for Query 2 under BASIC, untrained GREEDY, and trained GREEDY. Descriptions of the execution steps for each plan are given in Tables 3.5 to 3.7 respectively.
### 3.7. Evaluation

<table>
<thead>
<tr>
<th>Query Step</th>
<th>Description</th>
<th>Triple Pattern</th>
<th>Time (ms)</th>
<th>Variable Bindings Assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Retrieve GO cellular component annotations for Affymetrix probeset 53701 at</td>
<td>(probeset:53701 at, affy:xGeneOntology Cellular Component, ?.goComponent)</td>
<td>3,696</td>
<td>?goComponent =&gt; 1 binding</td>
</tr>
<tr>
<td>2</td>
<td>Retrieve GO biological process annotations for Affymetrix probeset 53701 at</td>
<td>(probeset:53701 at, affy:xGeneOntology Biological Process, ?.goComponent)</td>
<td>55</td>
<td>?goProcess =&gt; 1 binding</td>
</tr>
<tr>
<td>3</td>
<td>Retrieve ENSEMBL gene cross-references for Affymetrix probeset 53701 at</td>
<td>(probeset:53701 at, affy:xEnsembl, ?.geneID)</td>
<td>47</td>
<td>?geneID =&gt; 1 binding</td>
</tr>
<tr>
<td>4</td>
<td>Retrieve amino acid sequence variation records for all proteins in the UniProt database</td>
<td>(?annotation, rdf:type, core2:NaturalVariantAnnotation)</td>
<td>53,152</td>
<td>?annotation =&gt; 70,629 bindings</td>
</tr>
<tr>
<td>5</td>
<td>Retrieve GO molecular function annotations for Affymetrix probeset 53701 at</td>
<td>(probeset:53701 at, affy:xGeneOntology Molecular Function, ?.goFunction)</td>
<td>65</td>
<td>?goFunction =&gt; 1 binding</td>
</tr>
<tr>
<td>6</td>
<td>For each amino sequence variation record from Step 4, retrieve the associated SNP ID</td>
<td>(?annotation, rdfs:seeAlso, ?snpID)</td>
<td>TIMEOUT</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 3.5: Description of execution steps for Query 2 under BASIC, as depicted in Fig. 3.10a

<table>
<thead>
<tr>
<th>Query Step</th>
<th>Description</th>
<th>Triple Pattern</th>
<th>Time (ms)</th>
<th>Variable Bindings Assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Retrieve GO cellular component annotations for Affymetrix probeset 53701 at</td>
<td>(probeset:53701 at, affy:xGeneOntology Cellular Component, ?.goComponent)</td>
<td>2,968</td>
<td>?goComponent =&gt; 1 binding</td>
</tr>
<tr>
<td>2</td>
<td>Retrieve GO biological process annotations for Affymetrix probeset 53701 at</td>
<td>(probeset:53701 at, affy:xGeneOntology Biological Process, ?.goComponent)</td>
<td>44</td>
<td>?goProcess =&gt; 1 binding</td>
</tr>
<tr>
<td>3</td>
<td>Retrieve ENSEMBL gene cross-references for Affymetrix probeset 53701 at</td>
<td>(probeset:53701 at, affy:xEnsembl, ?.geneID)</td>
<td>48</td>
<td>?geneID =&gt; 1 binding</td>
</tr>
<tr>
<td>4</td>
<td>Retrieve amino acid sequence variation records for all proteins in the UniProt database</td>
<td>(?annotation, rdf:type, core2:NaturalVariantAnnotation)</td>
<td>54,759</td>
<td>?annotation =&gt; 70,629 bindings</td>
</tr>
<tr>
<td>5</td>
<td>Retrieve GO molecular function annotations for Affymetrix probeset 53701 at</td>
<td>(probeset:53701 at, affy:xGeneOntology Molecular Function, ?.goFunction)</td>
<td>65</td>
<td>?goFunction =&gt; 1 binding</td>
</tr>
<tr>
<td>6</td>
<td>Retrieve UniProt protein identifiers corresponding to Affymetrix probeset 53701 at</td>
<td>(probeset:53701 at, affy:xSwissProt, ?.proteinID)</td>
<td>42</td>
<td>?proteinID =&gt; 1 binding</td>
</tr>
<tr>
<td>7</td>
<td>Retrieve annotations for each protein from step 6</td>
<td>(?proteinID, core:annotation, ?annotation)</td>
<td>14,782</td>
<td>?proteinID =&gt; 1 binding ?annotation =&gt; 2 bindings</td>
</tr>
<tr>
<td>8</td>
<td>For each annotation from step 7, retrieve the corresponding SNP ID</td>
<td>(?annotation, rdfs:seeAlso, ?snpID)</td>
<td>1,997</td>
<td>?annotation =&gt; 2 bindings ?snpID =&gt; 2 bindings</td>
</tr>
</tbody>
</table>

Table 3.6: Description of execution steps for Query 2 under untrained GREEDY, as depicted in Fig. 3.10b
### 3.7. Evaluation

<table>
<thead>
<tr>
<th>Query Step</th>
<th>Description</th>
<th>Triple Pattern</th>
<th>Time (ms)</th>
<th>Variable Bindings</th>
<th>Assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Retrieve GO cellular component annotations for Affymetrix probeset 53701 at</td>
<td>(probeset:53701_at, affy:xGene_Ontology_Cellular_Component, ?goComponent)</td>
<td>2,951</td>
<td>?goComponent =&gt; 1 binding</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Retrieve GO biological process annotations for Affymetrix probeset 53701 at</td>
<td>(probeset:53701_at, affy:xGene_Ontology_Biological_Process, ?goComponent)</td>
<td>44</td>
<td>?goProcess =&gt; 1 binding</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Retrieve ENSEMBL gene cross-references for Affymetrix probeset 53701 at</td>
<td>(probeset:53701_at, affy:xEnsembl, ?geneID)</td>
<td>45</td>
<td>?geneID =&gt; 1 binding</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Retrieve GO molecular function annotations for Affymetrix probeset 53701 at</td>
<td>(probeset:53701_at, affy:xGene_Ontology_Molecular_Function, ?goFunction)</td>
<td>65</td>
<td>?goFunction =&gt; 1 binding</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Retrieve UniProt protein identifiers corresponding to Affymetrix probeset 53701 at</td>
<td>(probeset:53701_at, affy:xSwissProt, ?proteinID)</td>
<td>41</td>
<td>?proteinID =&gt; 1 binding</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Retrieve annotations for each protein from step 5</td>
<td>(?proteinID, core:annotation, ?annotation)</td>
<td>11,237</td>
<td>?proteinID =&gt; 1 binding and ?annotation =&gt; 2 bindings</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>For each annotation record from step 5, retrieve its record type</td>
<td>(?annotation, rdf:type, core2:Natural_Variant_Annotation)</td>
<td>14,247</td>
<td>?annotation =&gt; 2 bindings</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>For each Natural Variant Annotation record, retrieve the corresponding SNP ID</td>
<td>(?annotation, rdf:seqAlso, ?snpID)</td>
<td>165</td>
<td>?annotation =&gt; 2 bindings and ?snpID =&gt; 2 bindings</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.7: Description of execution steps for Query 2 under trained GREEDY, as depicted in Fig. 3.10c

In Figs. 3.10a to 3.10c, the reader will observe that many of the triple patterns are resolved very quickly. For instance, Steps 2, 3, and 5 of Fig. 3.10a are completed in 55, 47, and 65 milliseconds, respectively. This happens because the relevant data for these patterns has already been gathered by the service invocation for Step 1. In Step 1, SHARE resolves the pattern: (probeset:53701_at, affy:xGene_Ontology_Cellular_Component, ?goComponent) by issuing the following query to the Bio2RDF Affymetrix endpoint:

```
CONSTRUCT {
}
WHERE {
}
```

As a result, Step 1 retrieves not only triples with the affy:xGene_Ontology_Cellular_Component predicate, but all triples in the Affymetrix endpoint with probeset:53701_at as the subject. This includes all triples that match the query patterns of Steps 2, 3, and 5, and thus no additional service invocations need to be made in these steps. The query engine maintains a hash table to track which services have been invoked with which inputs, in order to avoid redundant service calls.

**Results for Query 4 (Fig. 3.8b)**

We see that for Query 4, there is no discernible difference between the performance of BASIC and GREEDY. The majority of query times are in the range 30-45 seconds, and this variation occurs across different trials for the same query plan, indicating that it is due to factors separate from the ordering of triple patterns (e.g. server load, network traffic). Query 4 contains no exceptionally expensive patterns, and there is no fan-out effect for any path of query traversal.
For example, there are only 2 Alzheimer’s-associated proteins in UniProt and only 2 known targets for the drug Varenicline. It is more efficient to solve the query by searching the protein-protein interactions of the Varenicline targets, as there is only 1 known interaction, whereas there are 58 known interactions for the Alzheimer’s proteins. However, there are relatively few orderings that would cause BASIC to use the latter strategy, and none of those orderings were included in the randomly generated test orderings.

**Results for Query 6 (Fig. 3.8c)**

The results graph for Query 6 shows similar patterns to the results graph for Query 4 (Fig. 3.8a). Here again, there are 3 distinct query times (∼ 140 seconds, ∼ 1300 seconds, and timeout), and the performance of each query depends on the handling of a single pattern:

\[
?\text{protein sadi:isParticipantIn GO:0006915}
\]

Similarly to `rdf:type` in Query 4, the `sadi:isParticipantIn` predicate is unusually expensive to resolve in the reverse direction. Using `GO:0006915` (apoptosis) as input, resolving `sadi:isParticipantIn` in the reverse direction requires approximately 20 minutes and produces 40,475 bindings for the `?protein` variable. It makes intuitive sense that this strategy would be expensive, because there may be a large number of proteins involved in a given biological process, particularly when proteins from all species are considered (as is the case here). In addition, retrieving the proteins associated with a given Gene Ontology term is more computationally intensive than a typical database lookup, because the results must include proteins that are annotated with any subterm (i.e., descendant term) of the target GO term. While these factors have a significant effect on query time, the factor that has the largest effect is the number of services that map to `sadi:hasParticipant` (the inverse predicate of `sadi:isParticipantIn`). In total, there are 27 services which map GO Terms to various types of database identifiers, and the query engine invokes each of these services sequentially.

As in Query 4, enabling GREEDY prevents query timeouts (e.g., Test Ordering 1 of Fig. 3.8c) that result from resolving a pattern with a large number of inputs. However, without training, GREEDY can also worsen query times in cases where the ordering of triple patterns is already efficient (e.g., Test Ordering 3 of Fig. 3.8c). Figs. 3.11a to 3.11c show the steps for Query Ordering 1 that lead to a timeout under BASIC (> 60 minutes), improved performance under GREEDY (∼ 20 minutes), and best performance under GREEDY with 2 training runs (∼ 2 minutes), respectively.

---

15 At the time of the experiment, many of the GO services (24 of 27) were not working correctly and were returning empty result sets. This did not affect the solutions for the query, as only one of the 27 services was actually relevant (GO ⇒ UniProt) in the overall context of the query. If all of the GO services had been working, the resolution of the `sadi:isParticipantIn` pattern would have been even more expensive.
3.7. Evaluation

(a) The steps followed by BASIC, in an attempt to answer Test Ordering 1 of Query 6 (Appendix B.2.3). The query times out after 60 minutes.

(b) The steps followed by GREEDY with 0 training runs, in order to answer Test Ordering 1 of Query 6 (Appendix B.2.3).

(c) The steps followed by GREEDY with 2 training runs, in order to answer Test Ordering 1 of Query 6 (Appendix B.2.3).

Figure 3.11: Execution plans for Query 6 under BASIC, untrained GREEDY, and trained GREEDY. Descriptions of the execution steps for each plan are given in Tables 3.8 to 3.10 respectively.
### 3.7. Evaluation

<table>
<thead>
<tr>
<th>Query Step</th>
<th>Description</th>
<th>Triple Pattern</th>
<th>Time (ms)</th>
<th>Variable Bindings Assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Retrieve proteins annotated with apoptosis or one of its subterms</td>
<td>(?protein, sadic:isParticipantIn, GO:0006915)</td>
<td>1,197,564</td>
<td>?protein =&gt; 40,475 bindings</td>
</tr>
<tr>
<td>2</td>
<td>Retrieve homologs for each protein from Step 1</td>
<td>(?protein, sadic:isHomologousTo, UniProt:Q93038)</td>
<td>TIMEOUT</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 3.8: Description of execution steps for Query 6 under BASIC, as depicted in Fig. 3.11a

<table>
<thead>
<tr>
<th>Query Step</th>
<th>Description</th>
<th>Triple Pattern</th>
<th>Time (ms)</th>
<th>Variable Bindings Assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Retrieve proteins annotated with apoptosis or one of its subterms</td>
<td>(?protein, sadic:isParticipantIn, GO:0006915)</td>
<td>1189735</td>
<td>?protein =&gt; 40,475 bindings</td>
</tr>
<tr>
<td>2</td>
<td>Retrieve homologs of UniProt:Q93038 (lymphocyte associated receptor of death)</td>
<td>(?protein, sadic:isHomologousTo, UniProt:Q93038)</td>
<td>56286</td>
<td>?protein =&gt; 23 bindings</td>
</tr>
<tr>
<td>3</td>
<td>For each protein from Step 2, retrieve the associated motifs</td>
<td>(?protein, sadic:hasMotif, ?motif)</td>
<td>27231</td>
<td>?protein =&gt; 23 bindings, motif =&gt; 4 bindings</td>
</tr>
</tbody>
</table>

Table 3.9: Description of execution steps for Query 6 under untrained GREEDY, as depicted in Fig. 3.11b

<table>
<thead>
<tr>
<th>Query Step</th>
<th>Description</th>
<th>Triple Pattern</th>
<th>Time (ms)</th>
<th>Variable Bindings Assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Retrieve homologs of UniProt:Q93038 (lymphocyte associated receptor of death)</td>
<td>(?protein, sadic:isHomologousTo, UniProt:Q93038)</td>
<td>63,426</td>
<td>protein =&gt; 475 bindings</td>
</tr>
<tr>
<td>2</td>
<td>For each homolog from Step 1, determine if it participates in apoptosis</td>
<td>(?protein, sadic:isParticipantIn, GO:0006915)</td>
<td>42,767</td>
<td>protein =&gt; 43 bindings</td>
</tr>
<tr>
<td>3</td>
<td>For each apoptosis protein from Step 2, retrieve the associated motifs</td>
<td>(?protein, sadic:hasMotif, ?motif)</td>
<td>22,709</td>
<td>protein =&gt; 43 bindings, motif =&gt; 4 bindings</td>
</tr>
</tbody>
</table>

Table 3.10: Description of execution steps for Query 6 under trained GREEDY, as depicted in Fig. 3.11c
3.8 Future Work

There are several areas where significant performance improvements to SHARE could be made, which are separate from the issue of ordering joins. Requests to different services are currently issued sequentially, whereas in many cases it is possible to issue requests in parallel. In addition, there are cases where calls to irrelevant services could be avoided, by taking the input and output datatypes of the services into account. For instance, the resolution of the sadi:isParticipantIn predicate in Query 6 leads to the invocation of 27 services which map GO terms to various types of database identifiers. However, only one of the GO services (GO \( \Rightarrow \) UniProt) is actually relevant in the overall context of the query. By matching the input/output datatypes of the different triple patterns, it should be possible to eliminate such unnecessary service calls. Another possible area for improvement is the pipelining of data. Most database systems pipeline data between query operators, in order to minimize the time-to-first-result, whereas SHARE determines the full solution set prior to displaying results. The fact that pipelining is not used in SHARE allows the joins to be freely reordered at any point in the query, which has been demonstrated in the experiment to be a significant advantage. However, it is possible that some compromise can be reached between pipelining and flexibility in reordering; for example, this same issue has been dealt with in the Tukwila system \[166\] by constructing query plans from pipelined “query fragments”.

3.9 Conclusion

The unique characteristics of the GREEDY algorithm are that: (1) it is adaptive, and (2) it can learn from previous queries. The evaluation of the optimizer demonstrates that while the GREEDY heuristic is successful in avoiding the worst execution plans, training is often necessary to get the best performance. In particular, the training aspect of the algorithm is important whenever there is a predicate that is significantly more expensive than average, given the same number of inputs. In the experiment, the two expensive predicates were rdf:type (when resolved in the reverse direction) and sadi:isParticipantIn (when resolved in the reverse direction). It is reasonable to handle rdf:type as a special case because it is part of the RDF standard, and the conventions surrounding its use are well established. However, there are likely to be many other types of relationships in the domain of bioinformatics, such as sadi:isParticipantIn, that must be treated specially in order to answer queries efficiently. For example, any predicates specifying broad criteria such as a target tissue or cellular location will likely have a high cost when resolved in the reverse direction. The training aspect of GREEDY provides a uniform and automatic way of dealing with such predicates that avoids manually coding special cases.

The experiment demonstrates that join ordering is an important factor for the performance of distributed queries. However, for queries that contain only a single constraint (i.e. only one triple pattern with a constant in the subject/object position), there is only one possible ordering of triple patterns, and thus the optimizer cannot produce any improvement. One of six queries in the evaluation was of this type (Query 3). At this point, it is difficult to say how common such queries will be under real usage of the system. A more ideal evaluation of the optimizer would sample real user queries from the logs of the SHARE demo, if/when the SHARE system is more widely used.

One interesting problem that is not addressed by GREEDY is the question of which query plan produces the most complete result set. Although the result sets for all queries in the experiment were verified to be the same, regardless of the execution plan, this is not guaranteed to be true for all SHARE queries in general. For example, cross references between biology databases are typically maintained by hand, and so the set of relationships between Database A and Database B might differ depending on which database is queried (i.e. which direction the predicate is resolved in). One idea for estimating the relative “completeness” of a predicate in the
forward and reverse directions would be to use a sampling technique. The system could sample predicate outputs from previous user queries and try to map them back to their original inputs, by resolving the same predicate in the opposite direction. In this way, a relative completeness score could be computed for the forward and reverse directions of each predicate. The scores could then be used to achieve a tradeoff between efficiency and completeness.

3.10 Source Code

The source code for SHARE is publicly available at http://sadi.googlecode.com and may be found in the sadi.share folder (i.e. http://sadi.googlecode.com/svn/trunk/sadi.share). In order to successfully build the project, the user must also check out the sadi.common and sadi.client projects. (The project build is automated with Maven.) The main procedure for the GREEDY algorithm is implemented in the executeQueryAdaptive method of the SHAREKnowledgeBase class (ca.wilkinsonlab.sadi.share.SHAREKnowledgeBase) and the logic for comparing the costs of query patterns is implemented in QueryPatternComparator, which is an inner class of SHAREKnowledgeBase. The code for recording, computing, and retrieving statistics about predicates may be found in the PredicateStatsDB class (ca.wilkinsonlab.sadi.stats.PredicateStatsDB).
Chapter 4

Conclusion

In the field of bioinformatics, making combined use of data sets and software packages often requires custom scripting work to reconcile incompatible formats, schemas, and interfaces. These incompatibilities exist because the various databases and programs are developed independently by different research laboratories, and without reference to a shared set of standards. Unfortunately, achieving standardization within the realm of bioinformatics is a daunting task due to the numerous, interrelated types of data (e.g., sequences, sequence annotations, molecular structures, chemical equations, molecular interactions, metabolic pathways, controlled vocabulary annotations) and analyses (e.g., gene prediction, multiple sequence alignment, protein structure prediction, subcellular location prediction, motif identification, phylogenetic tree construction) that exist. However, data/software integration work is mundane, time-consuming, and error prone, and it impedes scientific studies. The potential benefits of automated data integration in bioinformatics are widely recognized, as evidenced by the large number of research projects that have been devoted to solving the problem (e.g. [57, 118, 119, 132, 141, 145, 168, 169, 174, 175]).

In this thesis, we have described a distributed query system called SHARE, which aims to address the data and software integration problems that are currently prevalent in bioinformatics. However, a number of distributed query systems have already been developed for bioinformatics prior to SHARE, such as Kleisli [141], TAMBIS [168], BioMART [119], BioFlow [145], and BioMediator [174]. Moreover, all of these systems (including SHARE) are similar in their design; in each case, the system answers user queries by decomposing them into requests against data sources that support a common “wrapper” interface, as per the mediator architecture depicted in Fig. 1.7. The technical feasibility of these systems has been demonstrated repeatedly; however, none of the frameworks have been a success in the sense that they are widely used in bioinformatics work. In addition to the technical challenges, there is also a social aspect to the problem of integration that must be addressed. In particular, there are two important questions that need to be answered for any integration framework:

1. What incentive do data and software providers have for participating?

2. Will the framework last?

The main characteristic that distinguishes SHARE from other distributed query systems is that it is built using Semantic Web standards, and this difference has relevance for both of the questions above. Data and software providers that implement SADI services will not only be compatible with SHARE, but will also be compatible with a growing collection of tools [176] (e.g., triple stores and data browsers) and data sources (e.g., Bio2RDF [90], LODD [91], UniProt [11], Linked Life Data [177]) that are specifically designed to support data integration tasks. Further, SHARE has no special requirements for the encoding of RDF data that is consumed/generated by the data sources. Thus, providers may encode their RDF using any ontology that they wish, and the data can readily be used outside the context of SHARE.

While the use of Semantic Web technologies has benefits for data providers, the most significant benefits are related to the implementation of the framework itself. Typically, the creators of a mediator system must develop and maintain a mediated schema, and the wrappers for participating data sources must implement mappings to and from that schema. If the wrappers are implemented on the mediator side, as has been the case for many bioinformatics systems
(e.g. Kleisli, TAMBI, BioFlow, and BioMediator), this poses an additional burden of maintenance. In either case, the success of the framework depends upon the continuing diligence of its creators. Thus, it is a important advantage that SHARE requires no mediated schema or mappings. This is a unique consequence of using RDF as the medium for data exchange. Triple stores are capable of storing arbitrary graph structures, and thus when the system aggregates data during a query, there are no restrictions on the structure of the RDF data that is returned from each source. Of course, this unconstrained approach also has disadvantages. The effective “master” schema for SHARE is determined collectively by the data sources, and is likely to be far more chaotic than a centrally curated system. Moreover, meaningful integration will only be possible in cases where data sources use the shared identifiers for common entities and predicates.

The main contribution of this thesis is GREEDY, the query evaluation algorithm for SHARE that is described in Chapter 3. GREEDY differs significantly from the standard (i.e. System R style [160]) procedure that is used to process queries in relational databases, in order to address additional challenges that are present when answering queries across distributed resources. One of the main difficulties under the distributed scenario is that statistics about the data sources are usually not available. In a relational database, statistics such as the size of a table and the number of keys in an index are essential for formulating efficient query plans. The GREEDY algorithm offers two complimentary solutions for this problem. First, GREEDY employs an adaptive approach to query planning where one join is executed at a time, and future joins are selected based on the result sizes of previous joins. Second, GREEDY learns statistics about predicates as it executes a query, in order to improve the performance of future queries. One of the advantages of the GREEDY approach is that it is able to optimize queries over web services, rather than databases with a specific type of query interface [16]. As a result, the system is capable of learning statistics about arbitrary software. This is particularly valuable in the bioinformatics domain, where specialized software tools often play an essential role in data analyses, and where the available programs are just as diverse as the available data.

While SHARE is a promising prototype, there are number areas where further work is needed before it is ready for real use by bioinformaticians. One of the most important pieces of missing functionality is provenance tracking. Bioinformaticians must be able to review a “proof” of each solution which shows the chain of supporting facts (i.e. triples), and the particular services that asserted each of those facts. Another requirement for real world use will be fine-grained configuration of analytical tools. For example, if a “homolog” predicate is resolved by a BLAST service, users must be able to configure the parameters for the BLAST service, such as the probability cutoff, the seed size, the similarity matrix, and so on. However, it is not clear how to best support such configuration through the query interface. Moreover, it runs contrary to the basic purpose of SHARE, which is to enable users to ask questions without needing to know the locations and interfaces of the specific services that are used to answer them. Thus, SHARE may be best suited for simple, ad hoc queries such as “Which biological pathways involve both protein A and protein B?” or “Which proteins are most frequently targeted by breast cancer drugs?” In order to accomplish more sophisticated analyses (e.g. building a phylogenetic tree), a workflow composition tool such as Taverna [123] is more suitable. One potential method to combine the strengths of SHARE and Taverna would be to use SHARE for constructing template workflows, and Taverna for fine-grained configuration. Yet another area for further work will be developing user interfaces for SHARE. While the query interface to SHARE is powerful, it requires familiarity with the SPARQL query language, and is not well suited to exploration of the data. One promising avenue for obtaining new interfaces would be to adapt existing tools for exploring RDF data, such as RDF browsers [178, 179], graph browsers [180, 181], and faceted browsers [182, 183], to use SHARE as the underlying data source.

Recall from Section 2.3.2 that SPARQL endpoints are modelled as SADI services within the query engine.
Bibliography


[51] Markus Fischer, Quan K Thai, Melanie Grieb, and Jrgen Pleiss. DWARF - a data warehouse system for analyzing protein families. BMC Bioinformatics, 7(495), 2006. URL http://www.biomedcentral.com/1471-2105/7/495.


[58] SRS@EBI (srs.ebi.ac.uk), retrieved Feb 4, 2011. URL [http://srs.ebi.ac.uk/srsbin/cgi-bin/wgetz?-page+srsq2+-noSession](http://srs.ebi.ac.uk/srsbin/cgi-bin/wgetz?-page+srsq2+-noSession)


Bibliography


[82] World wide web consortium (W3C), retrieved Feb 4, 2011. URL http://www.w3.org/


[85] Turtle - terse RDF triple language, retrieved Feb 4, 2011. URL http://www.w3.org/TeamSubmission/turtle/

[87] LargeTripleStores - ESW wiki, retrieved Feb 4, 2011. URL http://esw.w3.org/topic/LargeTripleStores


83


[112] Web service definition language (WSDL), retrieved Feb 4, 2011. URL http://www.w3.org/TR/wSDL.


[116] dbfetch @ EBI, retrieved Feb 4, 2011. URL http://www.ebi.ac.uk/cgi-bin/emblfetch.

Bibliography


[146] Courtot Mélanie, Bug William, Gibson Frank, and et al. The OWL of biomedical investigations.


[156] SKOS simple knowledge organization system, retrieved Feb 4, 2011. URL [http://www.w3.org/2004/02/skos/](http://www.w3.org/2004/02/skos/).


Bibliography


Appendix A

Supporting Material for Chapter 2

A.1 An Example RDF Service Description for a SADI Service, Obtained by Performing an HTTP GET on the URL [http://sadiframework.org/examples/uniprot2pubmed](http://sadiframework.org/examples/uniprot2pubmed)

```xml
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:mygrid="http://www.mygrid.org.uk/mygrid-moby-service#">
    <mygrid:serviceDescription rdf:about="http://sadiframework.org/examples/uniprot2pubmed">
        <mygrid:hasServiceDescriptionText>Retruns PubMed ids associated with a UniProt record.</mygrid:hasServiceDescriptionText>
        <mygrid:hasServiceNameText rdf:datatype="http://www.w3.org/2001/XMLSchema#string">UniProt-to-Pubmed</mygrid:hasServiceNameText>
        <mygrid:hasOperation>
            <mygrid:operation rdf:about="http://sadiframework.org/examples/uniprot2pubmed#operation">
                <mygrid:outputParameter>
                    <mygrid:parameter rdf:about="http://sadiframework.org/examples/uniprot2pubmed#output">
                        <mygrid:objectType rdf:resource="http://sadiframework.org/examples/uniprot2pubmed.owl#AnnotatedUniProtRecord"/>
                    </mygrid:parameter>
                </mygrid:outputParameter>
                <mygrid:inputParameter>
                    <mygrid:parameter rdf:about="http://sadiframework.org/examples/uniprot2pubmed#input">
                        <mygrid:objectType rdf:resource="http://purl.oclc.org/SADI/LSRN/UniProt_Record"/>
                    </mygrid:parameter>
                </mygrid:inputParameter>
            </mygrid:operation>
        </mygrid:hasOperation>
    </mygrid:serviceDescription>
</rdf:RDF>
```
Appendix B

Supporting Material for Chapter 3

B.1 Variants of Training Queries Used to Evaluate the GREEDY Optimization Algorithm

B.1.1 Variants of Query 1

Variant 0):

```sparql
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX core2: <http://bio2rdf.org/core:>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX core: <http://purl.uniprot.org/core/>
PREFIX uniprot: <http://bio2rdf.org/uniprot:>

SELECT ?substitution ?start ?end ?dbCrossRef
WHERE {
  ?annotation rdfs:type core2:Natural_Variant_Annotation .
  ?annotation core:range ?range .
  ?range core:begin ?start .
  ?range core:end ?end .
  ?annotation rdf:type core2:Natural_Variant_Annotation .
  ?annotation rdfs:seeAlso ?dbCrossRef .
  ?annotation core:substitution ?substitution
}
```

Variant 1):

```sparql
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX core2: <http://bio2rdf.org/core:>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX core: <http://purl.uniprot.org/core/>
PREFIX uniprot: <http://bio2rdf.org/uniprot:>

SELECT ?substitution ?start ?end ?dbCrossRef
WHERE {
  ?annotation rdf:type core2:Natural_Variant_Annotation .
  ?annotation rdfs:seeAlso ?dbCrossRef .
  uniprot:Q9P126 core:annotation ?annotation .
  ?annotation core:substitution ?substitution .
  ?annotation core:range ?range .
  ?range core:begin ?start .
  ?range core:end ?end
}
```

Variant 2):

```sparql
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX core2: <http://bio2rdf.org/core:>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX core: <http://purl.uniprot.org/core/>
```
B.1. Variants of Training Queries

B.1.1 Variants of Training Query 1

Variant 3):

PREFIX uniprot: <http://bio2rdf.org/uniprot:>
SELECT ?substitution ?start ?end ?dbCrossRef
WHERE {
  ?annotation core:range ?range .
  ?annotation rdfs:seeAlso ?dbCrossRef .
  ?annotation rdf:type core2:Natural_Variant_Annotation .
  ?range core:end ?end .
  ?annotation core:substitution ?substitution .
  ?range core:begin ?start
}

Variant 3):

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX core2: <http://bio2rdf.org/core:>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX core: <http://purl.uniprot.org/core/>
PREFIX uniprot: <http://bio2rdf.org/uniprot:>
SELECT ?substitution ?start ?end ?dbCrossRef
WHERE {
  ?annotation core:range ?range .
  ?range core:begin ?start .
  ?annotation rdf:type core2:Natural_Variant_Annotation .
  ?annotation core:substitution ?substitution .
  ?range core:end ?end .
  ?annotation rdfs:seeAlso ?dbCrossRef
}

B.1.2 Variants of Query 3

Variant 0):

PREFIX diseasome: <http://www4.wiwiss.fu-berlin.de/diseasome/resource/diseasome/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX drugbank: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/>
PREFIX bio2rdf: <http://bio2rdf.org/ns/bio2rdf#>
PREFIX drug: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugs/>
SELECT ?targetProtein ?gene ?disease ?diseaseName
WHERE {
  ?disease rdfs:label ?diseaseName
}

Variant 1):

PREFIX diseasome: <http://www4.wiwiss.fu-berlin.de/diseasome/resource/diseasome/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX drugbank: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/>
PREFIX bio2rdf: <http://bio2rdf.org/ns/bio2rdf#>
PREFIX drug: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugs/>
SELECT ?targetProtein ?gene ?disease ?diseaseName
WHERE {
B.1. Variants of Training Queries

Variant 2):

```sparql
PREFIX diseasome: <http://www4.wiwiss.fu-berlin.de/diseasome/resource/diseasome/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX drugbank: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/>
PREFIX bio2rdf: <http://bio2rdf.org/ns/bio2rdf#>
PREFIX drug: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugs/>

SELECT ?targetProtein ?gene ?disease ?diseaseName
WHERE
{ drug:DB03881 drugbank:target ?targetProtein .
  ?disease rdfs:label ?diseaseName
}
```

Variant 3):

```sparql
PREFIX diseasome: <http://www4.wiwiss.fu-berlin.de/diseasome/resource/diseasome/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX drugbank: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/>
PREFIX bio2rdf: <http://bio2rdf.org/ns/bio2rdf#>
PREFIX drug: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugs/>

SELECT ?targetProtein ?gene ?disease ?diseaseName
WHERE
{ drug:DB04896 drugbank:target ?targetProtein .
  ?disease rdfs:label ?diseaseName
}
```

B.1.3 Variants of Query 5

Variant 0):

```sparql
PREFIX sadi: <http://sadiframework.org/ontologies/properties.owl#>
PREFIX ss: <http://semanticscience.org/resource/>
PREFIX GO: <http://lsrn.org/GO:>

SELECT ?protein ?omim
FROM <http://dev.biordf.net/~benv/properties.owl>
WHERE
{ ?protein sadi:hasFunction GO:0004872 . # "receptor activity"
  ?protein sadi:isParticipantIn GO:0007595 . # "lactation"
}
```

Variant 1):

```sparql
PREFIX sadi: <http://sadiframework.org/ontologies/properties.owl#>
PREFIX ss: <http://semanticscience.org/resource/>
PREFIX GO: <http://lsrn.org/GO:>

SELECT ?protein ?omim
FROM <http://dev.biordf.net/~benv/properties.owl>
```
B.2 Randomly Generated Orderings for Test Queries

Used to Evaluate the GREEDY Optimization Algorithm

B.2.1 Orderings for Query 2

Ordering 0):

WHERE
\{
?protein sadi:isParticipantIn GO:0048588 .
?protein sadi:hasFunction GO:0004872
\}

Variant 2):

PREFIX sadi: <http://sadiframework.org/ontologies/properties.owl#>
PREFIX ss: <http://semanticscience.org/resource/>
PREFIX GO: <http://lsrn.org/GO> 

SELECT ?protein ?omim 
FROM <http://dev.biordf.net/~benv/properties.owl> 
WHERE
\{
?protein sadi:isParticipantIn GO:0048588 .
?protein sadi:hasFunction GO:0004872
\}

Variant 3):

PREFIX sadi: <http://sadiframework.org/ontologies/properties.owl#>
PREFIX ss: <http://semanticscience.org/resource/>
PREFIX GO: <http://lsrn.org/GO> 

SELECT ?protein ?omim 
FROM <http://dev.biordf.net/~benv/properties.owl> 
WHERE
\{
?protein sadi:hasFunction GO:0004872 .
GO:0006260 sadi:hasParticipant ?protein .
?protein sadi:isCausallyRelatedWith ?omim
\}
B.2. Randomly Generated Orderings for Test Queries

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX core2: <http://bio2rdf.org/core:>
PREFIX affy: <http://bio2rdf.org/ns/affymetrix#>
PREFIX probeset: <http://bio2rdf.org/ns/affymetrix:>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX core: <http://purl.uniprot.org/core/>

WHERE
  ?annotation rdfs:seeAlso ?snpID .
  probeset:53701 a affy:xGene_Ontology_Cellular_Component ?goComponent
}

Ordering 2):

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX core2: <http://bio2rdf.org/core:>
PREFIX affy: <http://bio2rdf.org/ns/affymetrix#>
PREFIX probeset: <http://bio2rdf.org/ns/affymetrix:>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX core: <http://purl.uniprot.org/core/>

WHERE
  probeset:53701 a affy:xEnsembl ?geneID .
  ?annotation rdf:type core2:Natural_Variant_Annotation .
  ?annotation rdfs:seeAlso ?snpID .
  ?proteinID core:annotation ?annotation
}

Ordering 3):

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX core2: <http://bio2rdf.org/core:>
PREFIX affy: <http://bio2rdf.org/ns/affymetrix#>
PREFIX probeset: <http://bio2rdf.org/ns/affymetrix:>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX core: <http://purl.uniprot.org/core/>

WHERE
{ probeset:53701 a affy:xSwissProt ?proteinID .
  ?annotation rdf:type core2:Natural_Variant_Annotation .
  probeset:53701 a affy:xEnsembl ?geneID .
  ?annotation rdfs:seeAlso ?snpID .
  ?annotation rdfs:seeAlso ?snpID .
  ?proteinID core:annotation ?annotation
}

Ordering 4):
B.2. Randomly Generated Orderings for Test Queries

B.2.2 Orderings for Query 4

Ordering 0):

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX rdf: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX drugbank: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX core: <http://purl.uniprot.org/core/>
PREFIX omim: <http://bio2rdf.org/omim:>
PREFIX drug: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugs/>

SELECT ?targetProtein ?alzProtein ?alzProteinName
WHERE
{ ?alzProtein rdfs:seeAlso omim:104300 .
  ?alzProtein dc:title ?alzProteinName .
  ?target drugbank:swissprotId ?targetProtein .
  ?participant owl:sameAs ?alzProtein .
  ?targetProtein core:interaction ?interaction }
```

Ordering 1):

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX rdf: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX drugbank: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX core: <http://purl.uniprot.org/core/>
PREFIX omim: <http://bio2rdf.org/omim:>
PREFIX drug: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugs/>

SELECT ?targetProtein ?alzProtein ?alzProteinName
WHERE
  ?target drugbank:swissprotId ?targetProtein .
  ?alzProtein rdfs:seeAlso omim:104300 .
  ?participant owl:sameAs ?alzProtein .
  ?alzProtein dc:title ?alzProteinName }
```

Ordering 2):

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
```
B.2. Randomly Generated Orderings for Test Queries

SELECT ?targetProtein ?alzProtein ?alzProteinName
WHERE
{
  ?target drugbank:swissprotId ?targetProtein .
  ?alzProtein rdfs:seeAlso omim:104300 .
  ?alzProtein dc:title ?alzProteinName .
  ?participant owl:sameAs ?alzProtein
}

Ordering 3):

SELECT ?targetProtein ?alzProtein ?alzProteinName
WHERE
{
  ?alzProtein rdfs:seeAlso omim:104300 .
  ?participant owl:sameAs ?alzProtein .
  ?alzProtein dc:title ?alzProteinName .
  ?target drugbank:swissprotId ?targetProtein .
  ?targetProtein core:interaction ?interaction
}

Ordering 4):

SELECT ?targetProtein ?alzProtein ?alzProteinName
WHERE
{
  ?alzProtein rdfs:seeAlso omim:104300 .
  ?alzProtein dc:title ?alzProteinName .
  ?target drugbank:swissprotId ?targetProtein .
  ?participant owl:sameAs ?alzProtein .
  ?interaction core:participant ?participant
}
B.2.3 Orderings for Query 6

Ordering 0):

PREFIX UniProt: <http://lsrn.org/UniProt:>
PREFIX sadi: <http://sadiframework.org/ontologies/properties.owl#>
PREFIX GO: <http://lsrn.org/GO:>

SELECT DISTINCT ?motif
FROM <http://dev.biordf.net/~benv/properties.owl>
WHERE

   { ?protein sadi:isHomologousTo UniProt:Q93038 .
     GO:0006915 sadi:hasParticipant ?protein .
     ?protein sadi:hasMotif ?motif
   }

Ordering 1):

PREFIX UniProt: <http://lsrn.org/UniProt:>
PREFIX sadi: <http://sadiframework.org/ontologies/properties.owl#>
PREFIX GO: <http://lsrn.org/GO:>

SELECT DISTINCT ?motif
FROM <http://dev.biordf.net/~benv/properties.owl>
WHERE

   { ?protein sadi:isParticipantIn GO:0006915 .
     ?protein sadi:isHomologousTo UniProt:Q93038 .
     ?protein sadi:hasMotif ?motif
   }

Ordering 2):

PREFIX UniProt: <http://lsrn.org/UniProt:>
PREFIX sadi: <http://sadiframework.org/ontologies/properties.owl#>
PREFIX GO: <http://lsrn.org/GO:>

SELECT DISTINCT ?motif
FROM <http://dev.biordf.net/~benv/properties.owl>
WHERE

   { ?protein sadi:isHomologousTo UniProt:Q93038 .
     GO:0006915 sadi:hasParticipant ?protein
   }

Ordering 3):

PREFIX UniProt: <http://lsrn.org/UniProt:>
PREFIX sadi: <http://sadiframework.org/ontologies/properties.owl#>
PREFIX GO: <http://lsrn.org/GO:>

SELECT DISTINCT ?motif
FROM <http://dev.biordf.net/~benv/properties.owl>
WHERE

   { ?protein sadi:isHomologousTo UniProt:Q93038 .
     ?protein sadi:isParticipantIn GO:0006915
   }

Ordering 4):

PREFIX UniProt: <http://lsrn.org/UniProt:>
PREFIX sadi: <http://sadiframework.org/ontologies/properties.owl#>
PREFIX GO: <http://lsrn.org/GO:>

SELECT DISTINCT ?motif
FROM <http://dev.biordf.net/~benv/properties.owl>
WHERE
  { ?protein sadi:isHomologousTo UniProt:Q93038 .
    GO:0006915 sadi:hasParticipant ?protein .
    ?protein sadi:hasMotif ?motif
  }