BioDig: Architecture for Integrating Heterogeneous Biological Data Repositories Using Ontologies

by

Howard H. Chou

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

May 19, 2005 [illegible]

Copyright 2005 M.I.T.

Author________________________________

Department of Electrical Engineering and Computer Science
May 19, 2005

Certified by________________

C. Forbes Dewey Jr.
Thesis Supervisor

Accepted by________________________

Arthur C. Smith
Chairman, Department Committee on Graduate Theses

BioDig: Architecture for Integrating Heterogeneous Biological Data Repositories Using Ontologies

by

Howard H. Chou

Submitted to the
Department of Electrical Engineering and Computer Science

May 19, 2005

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

High-throughput experiments generate vast quantities of biological information that are stored in autonomous data repositories distributed across the World Wide Web. There exists a need to integrate information from multiple data repositories for the purposes of data mining; however, current methods of integration require a significant amount of manual work that is often tedious and time consuming. The thesis proposes a flexible architecture that facilitates the automation of data integration from multiple heterogeneous biological data repositories using ontologies. The design uses ontologies to resolve the semantic conflicts that usually hinder schema integration and searching for information. The architecture implemented successfully demonstrates how ontologies facilitate the automation of data integration from multiple data repositories. Nevertheless, many optimizations to increase the performance of the system were realized during the implementation of various components in the architecture and are described in the thesis.

Thesis Supervisor: C. Forbes Dewey Jr.
Title: Professor of Mechanical Engineering and BioEngineering, M.I.T.

Table of Contents

1.0 Introduction

2.0 Background

- 2.1 Data Repositories
- 2.2 Integration Methods
  - 2.2.1 Warehousing
  - 2.2.2 Multidatabase
  - 2.2.3 Federated
  - 2.2.4 Mediated
  - 2.2.5 Link
  - 2.2.6 Service
- 2.3 Biological Ontologies
- 2.4 Biological Identifiers

3.0 Design Goals

4.0 Technologies

- 4.1 Data Storage
- 4.2 Web Services
- 4.3 Ontologies
- 4.4 Unique Identifier

5.0 Design

- 5.1 Overview
- 5.2 User Interface

5.3 Query Processing

5.4 Results Aggregation

5.5 Ontology for Database Schema Integration

5.6 Ontology for Indexing Database Information

6.0 Architecture

6.1 Overview

6.2 User Interface

6.3 Schema Mapping

6.4 Query Processor

6.5 Results Aggregator

6.6 Data Cache

6.7 Data Storage

6.8 Remote Procedure Call (RPC)

6.9 Data Compression

6.10 LSID Resolver

7.0 Implementation

7.1 Hardware

7.2 Software

7.3 Biological Data Repositories

7.4 Biological Ontologies

7.5 Biological Unique Identifiers

7.6 Simplifications

7.7 Limitations

8.0 Discussion

8.1 Database Schema Integration

8.2 Query Processing

8.3 Results Aggregation

8.4 Semantic Web Application

9.0 Future Work

10.0 Conclusion

11.0 References

12.0 Appendix

I. QueryServlet.java

II. DbQuerySevice.jws

Table of Figures

Figure 1: Implementation of user interface for registering a new database. 38

Figure 2: Implementation of user interface for constructing a query. 39

Figure 3: Implementation of user interface for viewing query results. 41

Figure 4: Steps involved in translating a user query into sub-queries. 45

Figure 5: Overview of system architecture. 54

Figure 6: Database schema of metadata about registered data repositories and ontology 57

Figure 7: Database schema for data cache. 60

Figure 8: WSDL file for the web service located at the RPC module. 62

Figure 9: All of the software required to implement the system and their uses. 65

1.0 Introduction

Biology has shifted from a data poor to a data rich field during the past decade, following the completion of many sequencing projects and development of high-throughput experimental methods. The data generated are isolated in public databases that can be codified into either data banks or knowledge banks. Data banks, such as ArrayExpress [1] and the Yeast Resource Center Public Data Repository [2], are repositories for experimental data. Knowledge banks are repositories for annotations about biological molecules and systems. Some examples of knowledge banks include GenBank [3], the Protein Data Bank [4], and Swiss-Prot [5]. Despite the plethora of biological data available, most of the data repositories exist in isolation and are distributed at different sites on the World Wide Web. The autonomous nature of the data repositories creates many problems that hinder biologists from integrating information from multiple data sources.

There exists a need to integrate information from disparate data repositories, because each data repository addresses a specific biological problem or domain, and no single repository can store all of the biological data available. For example, given the primary sequence of a new protein, one might want to know whether any known human proteins have similar protein domains and are cancer related. One approach is to answering the question is to perform a BLAST [6] of the primary sequence and look through the proteins returned that are related to each of the conserved domains found in the sequence. Another approach is to identify potential protein domains using Prosite [7], and look through the proteins cross-referenced in each of the domains returned. Both methods can become extremely tedious, especially when there are a large number of proteins to manually search through. The query interfaces provided by public data repositories are web based and most of the cross-referenced information is provided as hyperlinks to other data repositories. Therefore, there is no easy way to automate the search process. Consequently, the wealth of information stored in the data repositories remain disconnected, making it difficult to mine information from multiple biological domains.

The difficulties associated with mining data from multiple data repositories are the diversity in the data, representational heterogeneity, autonomous and web-based nature, and different querying capabilities of the data sources [8]. Biological data are diverse and include sequence, structural, microarray, and interaction data. Each type of data is stored in various formats, such as flat files, XML files, and images. The data formats are often inconsistent between different data sources, and each data source operates autonomously in updating its information, which implies that data could easily become inconsistent across multiple data repositories. Furthermore, most query interfaces to the data sources are restrictive in the types of queries the user is allowed to pose. The user interfaces are also unique, so scientists are forced to learn a different interface for each data repository. These four problems make integrating information from multiple biological data sources an extremely challenging task.

Existing solutions for integrating multiple biological data sources differ along five dimensions: goal of integration, data model, source model, user model, and transparency [8]. The goal of integration describes the type of integration the system aims to support, such as navigational or query based. The data model refers to the type of data integrated, such as text, structured, or linked. The source model pertains to whether the data sources are complementary or overlapping. The user model describes the type of user for whom the system is designed, and the amount of expertise required to use the system. Finally, transparency refers to how much control a user has over the data sources used to answer his query. Design differences in each dimension have produced several integration systems, such as BACIIS [9], BioKleisli [10], BioNavigator [11], DiscoveryLink [12], Entrez [13], GUS [14], KIND [15], SRS [16], and TAMBIS [17]. Despite their differences along the different dimensions, most of the existing integration systems focus predominantly on the problem of representational heterogeneity.

The problem of representational heterogeneity can be separated into two sub-problems. The first sub-problem is a lack of a controlled vocabulary and consistent methods for representing biological data. Currently, the inconsistency in how biological terms are defined and the existence of multiple formats for representing the same type of experimental data greatly impede the integration of biological data. Recent work in the forms of standardizations, such as MIAME [18], and ontologies, such as the Gene Ontology (GO) [19] and MGED Ontology [20], begin to address both problems. However, data is often stored using different database schemas even when standards exist for how the data should be represented. The inconsistency makes it difficult for a machine to interpret a source schema when querying the database. Therefore, the second sub-problem is a lack of homogeneity in how data is represented in the databases.

The thesis focuses on the problem of representational heterogeneity in how data is represented in databases, and it provides a solution that aims to support a queryable system for integrating data from multiple heterogeneous databases. The solution consists of a distributed architecture that uses ontologies to facilitate the integration process. A lazy approach to integration [21] is used, because the queries are expected to be ad-hoc and operate over a large number of data repositories that could be changing rapidly. The lazy approach also has other advantages that are desirable. For example, data freshness is guaranteed, and the remote data repositories remain autonomous and transparent to the user. Furthermore, the distributed architecture is more conducive towards expandability, scalability, and portability. The system makes no assumptions about the source model and assumes rudimentary technical expertise from the user. The remaining sections look at some of the current architectures for integrating heterogeneous biological data repositories and describe a new solution for achieving the task that is posed.

2.0 Background

The following sections provide a brief overview of the different types of biological data repositories and existing methods for integrating them. The last two sections discuss some of the recent work related to the development of biological ontologies and identifiers.

2.1 Data Repositories

The number of biological data repositories has increased exponentially over the past decade to a current count of approximately 500 [8]. There are two types of data repositories: data banks and knowledge banks. Data banks contain experimental data stored in various formats, including tab-delimited flat files, XML files, images, and database entries. Knowledge banks contain annotated data stored mainly in databases, due to the large quantities of information involved and the need to search the information. Moreover, the scope of the data repositories can vary from addressing a specific biological problem to a particular biological domain. For example, a data bank might store all of the experimental data from a group of researchers testing a hypothesis, whereas a knowledge bank might contain all known information about protein crystal structures. As a result of the wide variation in the type of data stored, most of the data repositories remain isolated from one another. Some repositories attempt to weakly unify information by using cross-references to other repositories. For example, Prosite references Swiss-Prot accession numbers, and vice versa, in specifying the protein domains associated with each protein molecule. The query engine for a particular data repository is also specific to the data stored in that repository; consequently, the autonomy of the repositories makes it difficult to integrate data from multiple data repositories.

Although the rapid generation of biological data has helped to move the field of biology in new directions, it is important to start taking steps towards unifying the data in order to make them more accessible. For example, what if a biologist wants to identify all of the human genes and proteins that play a role in a specific part of a metabolic network, where the genes are regulated by a particular transcription factor and the proteins contain a particular protein domain. Currently, the biologist must search multiple data repositories and browse through many entries in each repository in order to find the subset of molecules that interest him, while keeping track of all of the information and cross-references from different repositories in his head. The search could easily become a time-intensive, complicated, and prohibitive task. However, the task could be simplified if a system for integrating information from multiple data repositories existed, because the system would reduce the amount of manual work the biologist is required to perform in order to find an answer to his question. Ideally, the biologist would submit his question to a service, and the service would answer the question by automatically searching the repositories that the biologist would have manually searched. By providing an aggregate view of the results from multiple data sources, it is possible that new insights, which were not apparent when the information from each data source was viewed in isolation, could be made.

2.2 Integration Methods

The following sections describe existing solutions to integrating heterogeneous biological data repositories and some of the systems that implement the different methods of integration. Existing methods fall into two main categories: lazy and eager [21]. In lazy integration, information is extracted from the data sources only when the system is presented with a query. Contrarily, information is extracted from the data sources and placed at a central repository in-advance during eager integration. The advantages and disadvantages of both systems are discussed below. Moreover, most of the current integration methods perform horizontal integration of biological data repositories, although some solutions are capable of weak vertical integration [22].

2.2.1 Warehousing

Data warehousing is the most common implementation of eager data integration, where information is integrated in-advance into a centralized repository that the user queries directly. In the warehousing approach, data is extracted from the repositories of interest, and the data is filtered, translated, and merged into a common database schema specified by the centralized repository. Therefore, data warehousing focuses more on data translation than query translation, since all queries are performed over a single database. Implementations of data warehouses for biological data include GUS [14], InterPro [23], and the BioWarehouse [24].

One major advantage of the warehousing approach is the absence of the problems associated with a distributed architecture, such as network bottlenecks, low response times, and unavailable data sources. Another advantage is the lack of inconsistency in how data are represented, because all of the data from the remote repositories are translated into a common format. However, the amount of data processing required to translate and merge the information from multiple repositories into a common database schema is extremely high and not something that should be repeated often. The high overhead cost associated with setting up a data warehouse makes it difficult to update information and add new repositories after the warehouse is set up. The issue of data freshness also arises, because the original data sources are not accessed to answer the query. Consequently, the freshness of the data depends on how often the data warehouse is rebuilt or updated.

Some of the properties usually considered as advantages of data warehousing are not necessarily advantages in the context of biological data. Filtered data might not be desirable, because partially incomplete or potentially incorrect data could be scientifically important. Data could also be lost during the filtering process; as a result, the data warehouse might lack the more detailed information contained in the original data sources, even though the warehouse contains a broader range of information. Although data warehouses could be optimized if the designer knows what portion of data would be accessed more frequently, queries to a biological data warehouse are often unpredictable and data could be accessed from anywhere in the database. Unfortunately, the unpredictable data access patterns make a biological data warehouse difficult to optimize.

2.2.2 Multidatabase

Multidatabase is the simplest implementation of lazy integration and provides integrated access to multiple relational database management systems (DBMSs). The multidatabase system achieves some transparency in data access by concealing how the component DBMSs are distributed, but the system does not hide the database schemas of the component DBMSs from the user. The multidatabase schema exposed to the user is the union of the component database schemas, where the tables' names are renamed with the database names appended as prefixes. Queries and updates are supported using two-phase commit. Query processing consists of decomposing the user query into sub-queries that could be answered by the component databases, and table joins could occur at both the component and multidatabase levels (joining information across multiple databases). An example of a multidatabase system is IBM's DataJoiner [25].

The lack of transparency is the main disadvantage for using multidatabase systems, because the user is required to be fluent in SQL and understand how the information is distributed amongst the databases in the integrated system in order to pose a query. The requirement that the component DBMSs must be relational DBMSs is another disadvantage for using multidatabase systems in the context of biological data, because the data model for biological data spans text, structured, and linked records. Furthermore, some structured data are stored in non-standard databases, such as object-oriented databases. The lack of metadata associated with the component databases complicates the automation of query processing and results aggregation, and limits the multidatabase system to performing horizontal integration.

2.2.3 Federated

The federated approach is similar to the multidatabase approach, except the component data sources can be any type of database, not just relational DBMSs. The federated system integrates the component databases with respect to both their schemas and data models, and information is aggregated on-demand. The federation is constructed from a series of mappings between views of the component databases that can be specified by the user or administrator of the federated database system (FDBS).

Abstractly, the FDBS is made up of three layers: data, export, and integration. The data layer consists of the component databases. Each database has an export schema that defines how the data are accessed and viewed by the outside world. Therefore, the export layer allows different data models to be unified consistently into the canonical data model defined by the integration layer. The integration layer is a unified view of the information available in the data layer based on the definitions and mappings in the export layer. The FDBS can be either loosely coupled or tightly coupled, depending on who defines the integrated schema. If the integrated schema is defined by the user, then the system is considered to be a loosely coupled FDBS. On the other hand, if the integrated schema is defined by the administrator of the FDBS, then the system is considered to be a tightly coupled FDBS.

Both types of FDBSs have their advantages and disadvantages with regards to transparency and autonomy. A tightly coupled FDBS has one integrated schema created by the FDBS administrator, who negotiates the structure of the export schemas with the administrators of the component databases. Therefore, the component databases have less autonomy, and changes to the source schemas usually require a new integrated schema to be generated. The advantage of a tightly coupled FDBS is that remote databases remain transparent to the user, because the administrator negotiates the query and update policies during the creation of the integrated schema. On the other hand, the component databases in a loosely coupled FDBS are more autonomous and less transparent. In a loosely coupled FDBS, users define their own integrated schema based on the export schemas provided by the administrators of the component databases. Therefore, the schemas of the component databases can be changed easily without breaking the integrated schemas. One disadvantage of a loosely coupled FDBS is that users must be exposed to the export layer and understand the structure of the export schemas in order to create their own integrated schemas. The existence of multiple integrated schemas could also result in the FDBS performing duplicate work, because users do not know about each other's schemas and are likely to create redundant definitions. Furthermore, updates are prohibited, because different users will require different update policies for their view of the component databases, which could create inconsistencies in the data during the update process.

Most implementations of the FDBS architecture consist of a federator that performs the integration task and source databases complemented by source-specific wrappers. A wrapper is a middle-ware application that supports the export schema by mapping the component data models to the canonical data model represented by the integrated schema. When the federator receives a new query, it first decomposes the query into sub-queries based on which databases are required to answer the query. Then, the sub-queries are transformed by inverting the mappings between the export and integrated schemas, so that the appropriate table and column names are used. Each sub-query is forwarded to the appropriate wrapper, which translates the sub-query into the query language and data model used by the underlying database. The results returned from the source database are sent to the wrapper, which translates the results so that they are consistent with the canonical data model. The federator aggregates the results returned from each wrapper, maps them back to the integrated schema, and returns the translated results to the user. Most FDBS systems are engineered in a bottom-up fashion, meaning that the set of databases that need to be integrated is known beforehand. The FDBS provides a single point of access to the integrated databases.

The major database systems capable of federation are Oracle Database [26] and IBM DB2 Universal Database [27]. Both database platforms perform synchronous integration by creating synonyms in the local database that refer to tables in the remote databases, and the user can incorporate remote tables into the local query by using those synonyms. The synonyms are usually created by the database administrator. One main difference between the two products is their views of virtualization, because IBM uses a shared-nothing architecture, while Oracle supports a shared-disk architecture [28]. Another implementation of the federated platform uses the ClassMapper approach in order to standardize how a database appears to the outside world [29, 30].

The advantages of a federated system are the support of ad-hoc queries, maintenance of data freshness, and the little additional storage required to support the FDBS. The disadvantages include many of those associated with lazy integration, such as long delays due to a slow network or low response times from the data repositories, overheads associated with translating between different source schemas and data models, interference of local queries by federated queries at the component databases, and unavailability of component databases. Furthermore, the user model in both Oracle and IBM DB2 require high levels of expertise to use and setup the FDBS, making the federation project costly.

2.2.5 Mediated

The architecture for the mediated system is similar to that of the federated system, except the mediated approach is designed to be more light-weight and flexible than the FDBS in order to support integration of information from non-database sources. However, the data sources are not as tightly linked as those in the federated system; as a result, the mediated architecture only supports read-only queries. The wrapper paradigm is still used in the mediated architecture, but the wrappers are also more complex, because non-database sources usually do not have source schemas defining how the underlying data is organized. In the mediated system, the mediator takes the place of the federator and performs query decomposition and data aggregation, and different implementations of the mediated architecture perform query decomposition differently. Unlike the federated system, mediated systems are usually engineered in a top-down fashion.

There are two main types of mediated systems: global-as-view (GAV) and local-as-view (LAV) [8]. The GAV approach represents the integrated schema as views over the source schemas, whereas the LAV approach represents the source schemas as views over the integrated schema. The advantage of GAV is that query decomposition is easy, because the relations in the integrated schema are already defined in terms of the relations in the source schema; however, the disadvantage is that adding and removing data sources to the mediator is hard, because the integrated schema must be updated to reflect the presence and absence of the data sources. Contrarily, query decomposition is much harder is LAV, but adding and removing data sources is easy.

One implementation of the mediated architecture for integrating multiple biological data repositories is IBM's DiscoveryLink [12]. DiscoveryLink is a middleware application that focuses heavily on query optimization over multiple data sources. The application acts as the integration layer by accepting a user query and communicating with the wrappers registered in the system to answer the query. The wrappers provide information about query capabilities and query execution costs of the underlying data sources, which the DiscoveryLink optimizer takes into account when constructing the overall query plan and deciding which data sources to access. Other mediated architectures include TAMBIS [17] and BioKleisli [10]. Although TAMBIS is an ontology-driven integration system, the ontology defined by TAMBIS is used primarily to represent relationships between various biological terms, not to integrate source schemas.

2.2.6 Link

Unlike the integration approaches mentioned thus far, link-based integration focuses on connecting hyperlinked biological information on the World Wide Web. Existing data repositories often provide information about the stored biological objects and cross-reference related objects at other repositories in the form of web pages. Users follow a point-and-click navigational process through a series of web pages in order to find the desired information; therefore, the navigational route followed is representative of a query. The link-based approach is based on the point-and-click paradigm and returns paths of links that lead to information relevant to a query. Consequently, the architecture of a link-based system is much simpler than those described earlier, because information is not physically integrated.

One implementation of the link-based approach is the Sequence Retrieval System (SRS) [16], which generates keyword indices on structured flat files and data banks that contain textual data. The cross references between data at different sources are also stored. A user searches for information using keywords, and the results of a query are a set of hyperlinked paths across different web pages that eventually lead to information relevant to the query. The user finds information by browsing the web pages associated with the hyperlinks returned. Other examples of link-based systems include BioNavigator [11] and Entrez [13].

The simplicity of the link-based system is its major advantage and disadvantage. The user model requires no critical expertise from the user, and setting up and updating the system consists of creating and updating an index. However, users are limited to keyword searches and are required to manually browse through multiple web pages in order to find the desired information. Furthermore, the success of the system is heavily dependent on the index, so it is possible for relevant information to be overlooked if the index is not constructed properly.

2.2.7 Service

Another approach to integrating biological data is to consistently bring together the services that access the data. Web services for retrieving and analyzing biological data already exist; however, it is difficult to connect the services, because most of them remain unknown to the general public and their input and output specifications are not published. Therefore, a services approach to integration registers the web services with a central repository and stores the input and output specifications for each registered service.

One implementation of a services approach to integrating biological data is the BioMOBY project [31]. The BioMOBY project seeks to create a central repository (MOBY Central) for biological web services and a language for describing the web services in terms of their inputs and outputs. The query process is similar to a workflow, where a user accesses different services registered with the central repository based on the data returned from previous queries until the desired information is found. Therefore, the user constructs his query in multiple steps, allowing him to explore multiple search paths. The exploration process can be time consuming, and most of the data integration is still performed by the user, because BioMOBY only facilitates the data integration process by directing the user to the services where he might find the information he needs. Other problems associated with the services integration design include the high probability that the central repository could become a bottleneck in the system, and the difficulties associated with describing the inputs and outputs of all web services in a consistent and succinct manner.

2.3 Biological Ontologies

Biology is a knowledge-based discipline, where knowledge is required for communication and formulating good predictions. Knowledge traditionally resided in the heads of experts, which was only feasible when experiments did not generate overwhelming quantities of data. However, after the recent shift in biology from a data poor to a data rich field, the knowledgebase has grown too large for any single person to assimilate. Many of the biological problems studied today also span multiple biological domains and require expert knowledge from multiple domains in order to solve the problems. Furthermore, knowledge that only resides in the heads of experts is not conducive towards the exchange of information. Therefore, a method for consistently organizing the knowledgebase to make its terms and relationships computationally available and tractable needs to be developed.

Bioinformaticists are exploring the uses of ontologies to represent the biological knowledgebase, due to the success of the artificial intelligence (AI) community in deploying ontologies that provide AI applications with domain knowledge. Uschold et al. describe an ontology as "tak[ing] a variety of forms, but necessarily it will include a vocabulary of terms, and some specification of their meaning. This includes definitions and an indication of how concepts are inter-related which collectively impose a structure on the domain and constrain the possible interpretations of terms" [32]. Therefore, the goal of a biological ontology should be to provide a controlled vocabulary that consistently represents the terms and semantic relationships in a particular biological domain.

There are three types of ontologies: domain-oriented, task-oriented, and generic; most biological ontologies are a mixture of the three [33]. Some of the biological ontologies developed can be found on the Open Biological Ontologies (OBO) website [34], such as the Gene Ontology (GO) and the NCI Thesaurus (NCIt) [35]. Other ontologies not found on the OBO website include the Ontology for Molecular Biology (MBO) [36], the RiboWeb Ontology [37, 38], and the EcoCyc Ontology [39]. The existing ontologies often differ in their representation of conceptual relationships and constraints, because the structure and content of an ontology is determined primarily by the tasks that the ontology is designed to facilitate. Therefore, it is possible for different ontologies in the same domain to present disparate views of the same underlying knowledge.

2.4 Biological Identifiers

Another major problem associated with the exchange of biological information is the inconsistency in naming biological objects. Most data repositories create their own unique identifiers for each object, and the format of the identifiers is completely arbitrary. Therefore, it is possible for multiple different biological objects from different repositories to be associated with the same identifier, and it is also possible for one biological object to be referred to by multiple different identifiers. The absence of globally unique identifiers for biological objects greatly hinders the exchange and access of biological information.

Two methods for implementing globally unique biological identifiers are to have a single centralized authority that distributes identifiers or to use a standard format for creating identifiers. A centralized repository could establish a one-to-one mapping between biological objects and their identifiers by tracking every biological object created in the knowledgebase. However, biology is currently changing too rapidly for any feasible implementation of a centralized repository, because names are constantly coming and going out of style, making it extremely difficult for a single authority to maintain an accurate mapping between biological objects and their unique identifiers. The alternative solution is to use a standard format for creating identifiers to ensure that every identifier generated is globally unique, and to distribute the work of handing out identifiers by creating different authorities for different domains. One standard developed is the Life Science Identifier (LSID) [40]. An LSID is a Universal Resource Name (URN) that syntactically identifies the source where the identifier came from and the biological object the identifier is associated with in the context of that source. Similar to a Domain Name Server (DNS), a LSID resolver is required to retrieve the biological object associated with a LSID, because users cannot find the location of an object simply from the LSID. However, the LSID system does not prevent a single biological object from having multiple different LSIDs.

The absence of globally unique identifiers for biological objects is both a technical and social problem. Engineers are developing methods to prevent identifiers from clashing; however, inconsistencies in the naming of biological objects will persist if scientists continue to refuse to adopt a universally consistent method for naming biological objects. Furthermore, identifiers need to be reused when referring to the same biological object in different domains in order for a one-to-one mapping to be created between an object and an identifier. The reuse of identifiers cannot be achieved until scientists are willing to adopt a standard format for describing biological identifiers.

3.0 Design Goals

The goal of the thesis is to build a prototype system that tackles the problem of representational heterogeneity when integrating heterogeneous, autonomous biological databases. Existing architectures provide partial solutions that are often inappropriate for the dynamic nature of biological data sources. For example, data warehouses have high overhead costs associated with setting up and updating the warehouse, which is disadvantageous when the number of new data sources is constantly growing. The multidatabase approach lacks location transparency and requires that the user know where the desired information is located, implying that the user must be knowledgeable about what types of information each data source in the multidatabase contains. Furthermore, the federated architecture is usually built bottom-up, so the set of data sources that are integrated must be known beforehand. The LAV (local-as-view) mediated architecture is the most suitable solution for integrating autonomous, heterogeneous biological data sources; however, most implementations lack a flexible architecture for doing schema integration that accounts for the semantic information embedded within the source schemas.

The thesis proposes an architecture that uses ontologies to facilitate semantic integration of information from autonomous, heterogeneous databases. Other desirable properties designed into the system include expandability, scalability, and usability. The system should be expandable to accommodate the addition of new databases with little additional overhead. The requirement on expandability minimizes the barrier to entry, so it is easy to register new databases as additional sources of information to search. A low barrier to entry would also allow the system to grow rapidly and quickly increase the quantity of searchable biological information. However, to prevent performance degradation from occurring as more databases are added, it is also essential that the system scales well. It is expected that many databases will be registered as the system grows, which will attract more users to the system. Therefore, the system should be responsive and scale appropriately as the number of remote databases and the number of users increase. Scalability can be achieved in part by ensuring that the databases remain autonomous, so that each remote database is maintained and updated independently. Site independence reduces the maintenance cost at multiple ends and promotes transparency, so users do not need to worry about what happens when remote databases become temporarily unavailable. The system should also be portable to facilitate expandability on different hardware and software architectures. Although the freshness of data is important, the requirement might be relaxed in favor of better performance. Most importantly, the system must be usable to biologists, who might have little technical background, so the learning curve for using the system should be low.

4.0 Technology

The following sections discuss the four major technologies required to build the proposed architecture for integrating and querying autonomous, heterogeneous biological databases.

4.1 Data Storage

The data storage layer is the heart of the system, because there would be no application if there were no information to integrate. Data sources store various types of information, such as experimental data, annotations, ontologies, and metadata; consequently, the data source must support an extremely diverse data model, since no standards for storing biological information currently exist. The data are currently stored in a wide variety of data formats including structured text, unstructured text, Excel spreadsheets, XML files, XML databases, relational database entries, and object-relational database entries. Although each format could potentially be supported in the data storage layer, the architecture designed in the thesis focuses on integrating information from relational databases.

The thesis focuses on relational databases, because the relational database is a well established platform for data storage and already implements many of the properties that are desirable in a data storage layer. The relational database is more flexible than text, Excel spreadsheets, and XML, because it supports a diverse data model, has a well defined query language, and contains different levels of access control. Furthermore, most relational databases are ACID (Atomic

Consistent Isolated Durable) compliant, so the integrity and consistency of the data returned could be trusted. There also exists a community of developers with broad expertise in using relational databases and programming in SQL, who could integrate existing biological web services consisting of relational database backends with the architecture proposed in the thesis. Although the object-relational database claims to be more powerful than a relational database due to its ability to define new data types, many of the extensions in existing implementations are still being defined and the advantages of the object-relational model have yet to be proven. Most importantly, the majority of existing biological data sources are relational database entries or structured text that could be translated into relational database entries. Therefore, the relational database is a good platform for data storage to focus on integrating first.

4.2 Web Services

The communication layer of the system is comprised of web services. A web service iany service that is available over the Internet, uses a standardized XML messaging system, and is not tied to any one operating system or programming language [41]. Web services are essential in application-centric environments, where distributed machines must cooperate in order to accomplish a common task. Cooperation requires communication, and the XML messaging systems provide standard protocols through which different machines can exchange information. Examples of XML messaging systems include XML documents, XML Remote Procedure Calls (XML-RPC) [42], and Simple Object Access Protocol (SOAP) [43]. Web services integration projects, such as BioMoby and MyGrid [44], attempt to integrate various existing web services related to biology in order to make them more accessible to the user.

Web services are important to biology, because it is impossible for a single site to have the resources and expertise required to construct and maintain all of the computational tools required by biologists. Existing services, such as the Distributed Annotation System (DAS) [45], are distributed across the World Wide Web and maintained by different groups who are experts in their domains. Despite the fact that new services are constantly being created, the exchange of information between new and old services is greatly facilitated by the message exchange protocols, because the input and output formats adhere to standard specifications. Web services are a key component of the communication layer in the system designed in the thesis, because of the architecture's distributed nature – different machines could host different data sources or perform various tasks, including query translation, query processing, query redirection, and results aggregation.

The architecture designed mimics the distributed nature of biological data repositories on the World Wide Web. However, the disconnectedness of the data repositories requires a method of communication to bridge the physical gap in order to aggregate the appropriate information necessary to answer a user's query. Web services provide a light-weight, flexible, and consistent method of communication between multiple machines.

4.3 Ontologies

The system designed uses ontologies to tackle the problems associated with representational heterogeneity in source schemas and data models. Representational heterogeneity in source schemas hinders the automated integration of information from multiple data sources, because schema integration is more complicated than just taking the union of different source schemas. Data sources often contain both complementary and overlapping information, which must be communicated to the system in order for it to correctly integrate information from multiple data repositories. Furthermore, a method for indexing information stored in the data repositories, which range from textual to binary data, is necessary to facilitate searching for information, because pattern matching is not semantically accurate, searching the same piece of text multiple times for various terms is not efficient, and no existing methods for searching binary data exists. Therefore, a method for representing the semantic information embedded in source schemas and a new approach for indexing data are necessary to facilitate searching and integrating information from heterogeneous data repositories.

An ontology could facilitate the search and integration process by providing a controlled vocabulary that consistently represents the semantic terms and relationships of a particular domain. A controlled vocabulary is a set of semantically defined terms, where the definitions are explicitly stated and agreed upon by a group of people. One prevalent problem in biology is that biological terms often have different definitions in the context of different biological domains; hence, pattern recognition techniques that naively match words are insufficient for searching textual information. However, queries could be semantically meaningful if the information in the data sources were indexed by ontological terms and the user incorporated those terms into their queries, since ontological terms are semantically meaningful keywords. Therefore, the machine can increase the precision without necessarily decreasing the recall of the search by knowing which connotation of the query term to use based on the associated ontology.

Semantically meaningful keywords are also useful as metadata for information that cannot be searched with existing technologies, such as binary data. The machine can determine whether the binary data is relevant to a user's query based on the ontological terms associated with the data. The inconsistency in how terms are used is also observed in biological data source schemas, because the names of database tables and columns are seldom unique. The inconsistency makes schema integration another extremely difficult problem.

Schema integration would be much easier if machines could understand whether two columns with the same name are composed of different data types and whether two columns with different names are composed of the same data type. Similar to biological terms, column names in source schemas are usually not associated with explicit definitions. As a result, it is possible for different column names in different schemas to represent the same biological object and the same column name in different schemas to represent different biological objects. Moreover, most existing efforts at data integration from heterogeneous data sources only succeed at horizontal integration, because it is difficult for the machine to determine whether data from different sources are semantically similar or not. Vertical integration requires that the machine is capable of differentiating between semantically similar and different data. Therefore, a set of semantically defined terms, such as an ontology, could alleviate the problem by mapping semantically equivalent columns to the same ontological term and semantically different columns to different ontological terms. The mappings could be used to automatically determine whether the data from different columns in different data repositories are semantically similar enough to integrate.

4.4 Unique Identifiers

The distributed nature of the designed architecture requires that the system be interoperable. Interoperability could be achieved in part by enforcing consistency in the identifiers associated with data from different databases. Therefore, the system designed in the thesis uses globally unique identifiers in order to facilitate the access and sharing of information from multiple databases. The thesis deals with the technical problem associated with unique identifiers, using LSIDs as globally unique biological identifiers and the BioPathways Consortium's LSID resolver client [46] as the LSID resolver.

Globally unique identifiers are essential for minimizing confusion about what data to retrieve and the location from which to retrieve the data. It is often the case that one biological entity is referred to by multiple identifiers, where each identifier is unique to the data repository that generates it. For example, the protein identified as Q43495 in Swiss-Prot is the same protein identified as IPR003612 in InterPro. The existence of multiple identifiers that refer to the same biological object creates confusion, because one can never be sure whether two different identifiers from two different data repositories actually refer to the same biological object without looking up both identifiers. It becomes even more confusing when one would like to determine the number of unique identifiers present amongst multiple identifiers from multiple data repositories, as in the case of data aggregation from multiple data repositories. If the same identifier is used in multiple repositories to refer to different biological objects, then it is impossible for a machine to determine which object the user wants to retrieve based only on an identifier. Therefore, global unique identifiers are required for uniquely identifying biological entities, so that information about the same biological object refers to the same unique identifier, which would greatly facilitates data integration and data access. Nevertheless, unique identifiers do not guarantee a one to one mapping between an identifier and a biological object, as mentioned in the Background. A one to one mapping between identifiers and objects would also require the adoption of a standard system for generating unique identifiers by the biological community.

5.0 Design

The system can be divided into three layers: user interface, query processing, and results aggregation. The first section provides a general overview of the system and describes design decisions that influence the overall architecture of the system. The latter sections detail the design of the user interface, how queries are processed into sub-queries that can be answered by the remote databases, and how results from the remote databases are aggregated. The last two sections discuss the roles of ontologies in the integration process.

5.1 Overview

The following issues related to integrating data from multiple databases were deemed important based on the designs of existing solutions to integration: goal of integration, data model, source model, user model, query model, transparency, bottom-up vs. top-down, virtual vs. materialized, read-only vs. read-write, and tight vs. loose integration. The system designed in the thesis does not focus on all of the issues, because some of them have already been successfully addressed by existing solutions. The Future Work section discusses how the solution presented in the thesis could be combined with some of the existing solutions to create an even better system for integrating data from multiple data sources.

The thesis focuses on the design of a queryable system that uses ontologies to facilitate data integration from multiple autonomous, heterogeneous databases. The goal of integration is to be able to query the information located at the remote databases. No assumptions are made on the source model of the integrated system, because ontologies are used to facilitate both horizontal and vertical integration. Therefore, the information in the databases could be complementary or overlapping. The query model is limited to structured queries in SQL, since most relational DBMSs support SQL as a query language. However, the system strives to achieve a user model that does not require much expertise to use, so a query rewriting module is incorporated into the system in order to prevent the user from having to write SQL and to allow the remote databases to be fully transparent to the user. The user interacts with the fully federated schema through the ontology that is mapped to the columns in the source schemas. The system is engineered using a top-down approach and performs virtual integration of data from loosely coupled databases that provide read-only access.

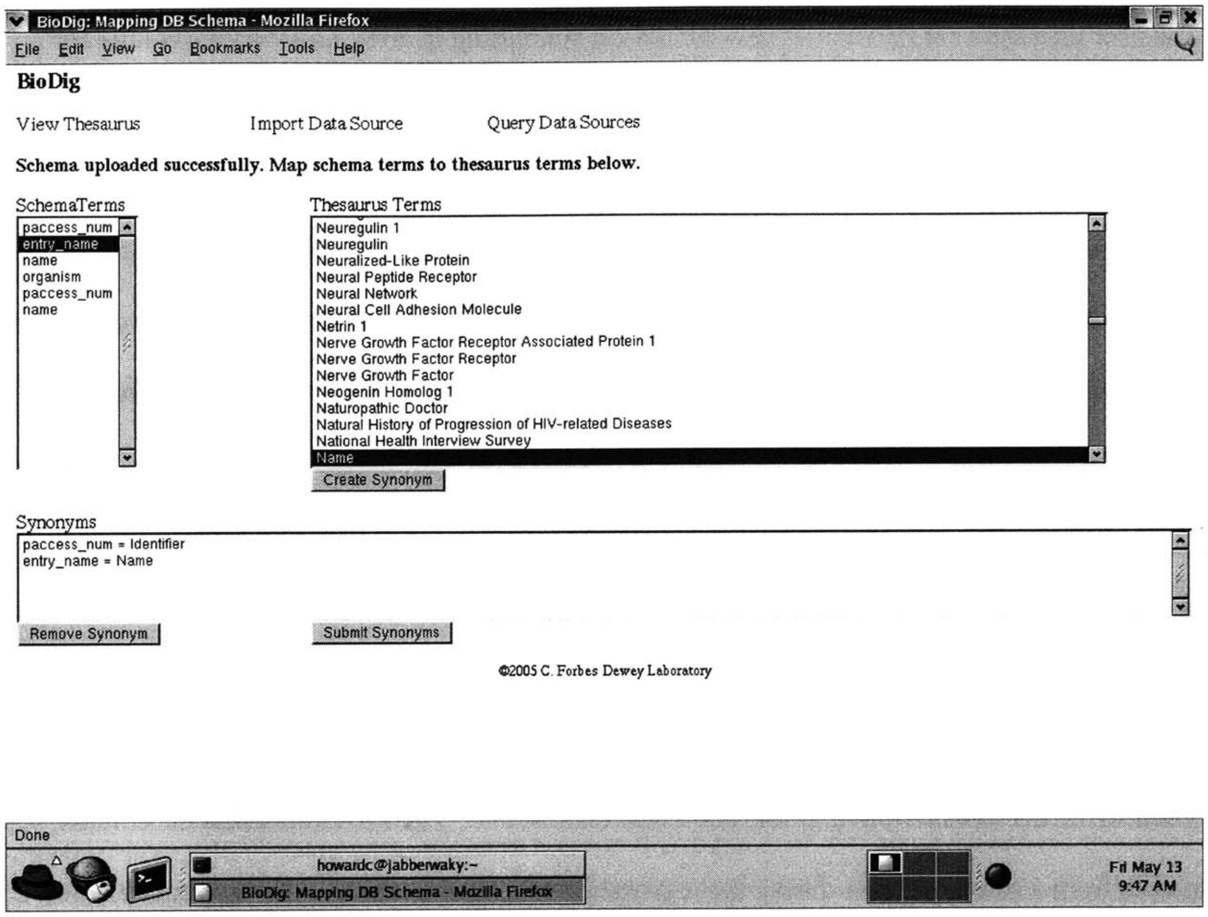
A distributed architecture is required to support virtual integration in order to add and remove remote databases easily. Virtual integration is preferred over materialized integration (data warehousing), because the system must be able to dynamically add new databases with little additional overhead as new data and knowledge banks are created. Furthermore, data freshness is easier to maintain with virtual integration, since data is retrieved from the data sources during query processing. Data freshness is important for presenting the user with an up-to-date and accurate view of information. A top-down approach is taken instead of a bottom-up approach, because it is impossible to predict the set of databases that should be integrated beforehand, and it is highly probable that new databases will be added to the system in the future. Although a bottom-up approach is easier to optimize, the optimizations make it difficult to add new databases and require that the entire system be rebuilt every time a new database is added. Flexibility in the system is also maintained by keeping the databases loosely coupled. One major drawback of a loosely coupled system is the difficulty involved in supporting read-write access, because each database has its own update policy. Therefore, the system only supports read-only access to the remote databases, which perform updates independently.

Integration of data from remote databases is facilitated by the use of ontologies. An ontology provides a consistent semantic vocabulary for defining the source schemas and describing the information stored in the databases. However, experts from different domains might not agree with each other on the semantic definitions for various biological terms, which impacts how data is integrated. Therefore, the system provides users with the flexibility to incorporate ontologies that they define for integrating data across multiple databases in order to accommodate different perspectives on the same biological information.

5.2 User Interface

The user interfaces for registering new databases, submitting queries, and viewing results are designed to require little expertise to use, adhering to the user model proposed in the design goals. The simplicity is achieved by using a loosely integrated architecture that maintains full transparency of the remote databases. The loose coupling between the remote databases allows users to define their own views of the knowledgebase based on the mappings that they create between the source schemas of the integrated databases and the ontology used to integrate the databases.

A new database is registered with the system by mapping the columns in the source schema to the terms in the ontology that represents the user's semantic view of a particular domain. The source schema consists of the SQL commands used to create the tables in the database, minimizing the work required to register a new data source. The system extracts the column names from a source schema during the registration of a new database. The user interface for creating the mapping consists of two columns: one column lists the column names from the newly loaded source schema, and the other column lists the names of the ontological terms with which the column names could be associated. Figure 1 shows an implementation of the interface described for registering a new database. A new mapping is created by choosing a column name and the ontological term to associate with the name. The new mappings are listed as they are created, so that the user can delete any mistakes in the mappings before they are registered. The new database is registered once the mappings from its schema to the ontology are created, and the mappings are used during query processing for determining which databases need to receive sub-queries in order to answer the user's query.



Schema uploaded successfully. Map schema terms to thesaurus terms below.

SchemaTerms
paccess_num
entry_name
name
organism
paccess_num
name

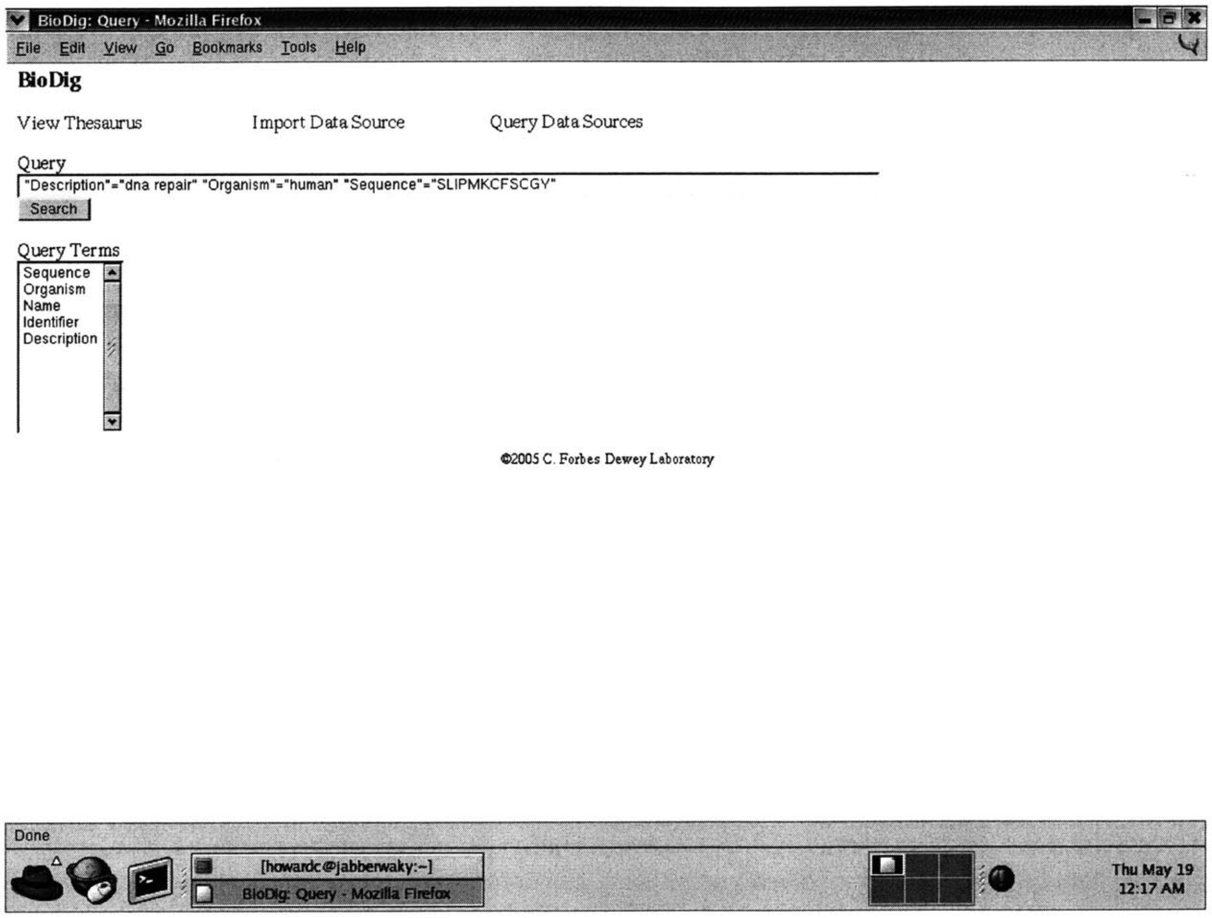
Thesaurus Terms
Neuregulin 1
Neuregulin
Neuralized-Like Protein
Neural Peptide Receptor
Neural Network
Neural Cell Adhesion Molecule
Netrin 1
Nerve Growth Factor Receptor Associated Protein 1
Nerve Growth Factor Receptor
Nerve Growth Factor
Neogenin Homolog 1
Naturopathic Doctor
Natural History of Progression of HIV-related Diseases
National Health Interview Survey
Name

Synonyms
paccess_num = Identifier
entry_name = Name

©2005 C. Forbes Dewey Laboratory

Figure 1: Implementation of user interface for registering a new database.

Queries are treated as filters of information in the knowledgebase, so query formulation is designed to facilitate the filtering process. Each query term is treated as a filter for information, and users are permitted to string together as many query terms as they deem necessary to express the question they want answered. A query term consists of an ontological term and one or more filter terms. For example, if the user wants all proteins with a specific sequence, then the filter term would be the desired sequence. Therefore, the user selects the ontological terms from the list of terms that have been mapped to database columns and specify what keywords to associate with each term in order to construct a query. An implementation of the interface for how a query is constructed is shown in Figure 2. The query is different from a naïve keyword search, because the ontology adds semantic meaning to the search, so that it is not a blind attempt at doing pattern matching.

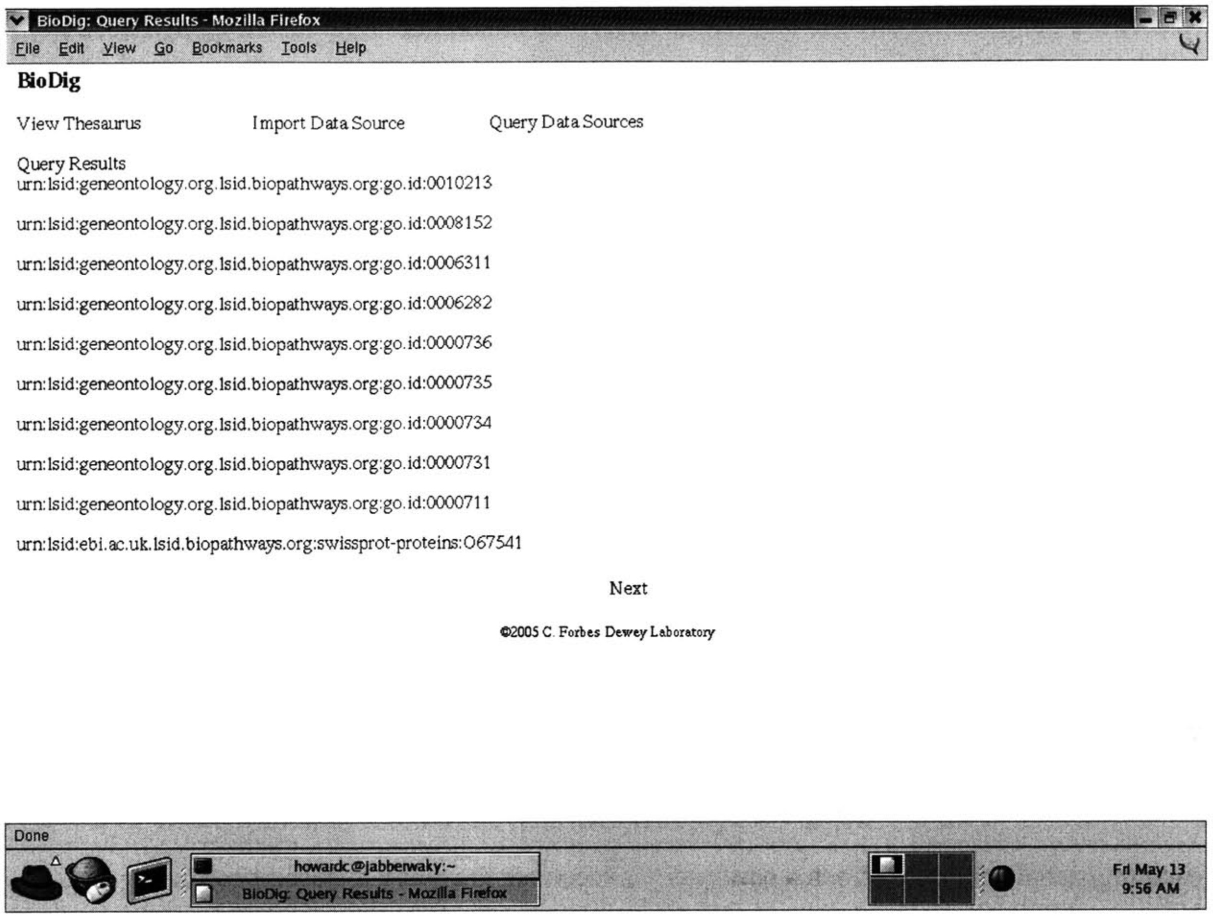


"Description"="dna repair" "Organism"="human" "Sequence"="SLIPMKCFSCGY"

©2005 C. Forbes Dewey Laboratory

Figure 2: Implementation of user interface for constructing a query.

The results from a query are returned to the user as LSIDs. A LSID resembles a uniform resource locator (URL) and is a compact representation of a biological object. Returning an identifier as a result is more efficient than returning the entire biological object, because the load on the network is significantly less; however, LSID resolvers are required to locate the biological object associated with each LSID. The LSID returned as a result to the user is the identifier for the biological object that best matches the user's query. The significance of the result is determined by the number of times the corresponding biological identifier is returned from multiple databases during aggregation. For example, if five databases were queried and the same identifier is returned from all five databases, then the object associated with that identifier is probably a significant result. The LSIDs are returned as hyperlinks to the LSID resolvers that can resolve the identifier and listed in order of significance. Users retrieve information by clicking on the hyperlinks. Figure 3 shows an implementation of the interface for viewing the results of a query.


BioDig: Query Results - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
BioDig
View Thesaurus
Import Data Source
Query Data Sources
Next
©2005 C. Forbes Dewey Laboratory
Done
howardc@jabberwaky:~
BioDig: Query Results - Mozilla Firefox
Fri May 13
9:56 AM

Query Results
urn:lsid:geneontology.org.lsid.biopathways.org:go.id:0010213
urn:lsid:geneontology.org.lsid.biopathways.org:go.id:0008152
urn:lsid:geneontology.org.lsid.biopathways.org:go.id:0006311
urn:lsid:geneontology.org.lsid.biopathways.org:go.id:0006282
urn:lsid:geneontology.org.lsid.biopathways.org:go.id:0000736
urn:lsid:geneontology.org.lsid.biopathways.org:go.id:0000735
urn:lsid:geneontology.org.lsid.biopathways.org:go.id:0000734
urn:lsid:geneontology.org.lsid.biopathways.org:go.id:0000731
urn:lsid:geneontology.org.lsid.biopathways.org:go.id:0000711
urn:lsid:ebi.ac.uk.lsid.biopathways.org:swissprot-proteins:O67541

Figure 3: Implementation of user interface for viewing query results.

5.3 Query Rewriting

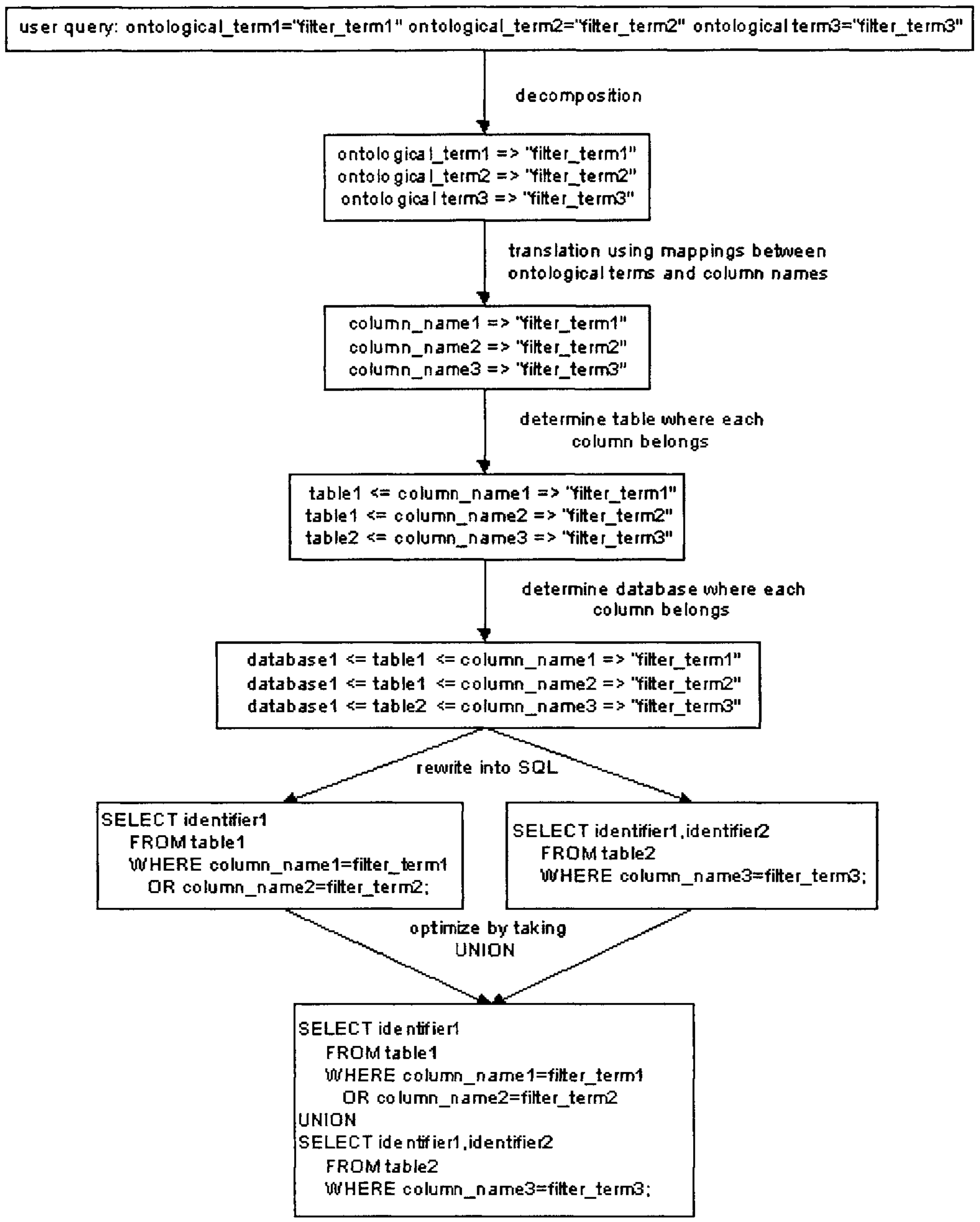
The simplicity of the user interface for constructing queries requires a more complicated query rewriting scheme in order to translate the user query into the appropriate SQL queries that could be answered by the component relational DBSs. There are three essential steps in reformulating a user query: translating the ontological terms into the appropriate column names, determining which databases need to be queried, and redefining the query into sub-queries based on which remote databases are required to answer the query. The mappings created during the registration of new databases play a key role in the query rewriting process.

The first step in translating the user query involves reverse mapping the ontological terms to the associated database column names. During the registration of a new database, column names are mapped to the ontological terms with which they share the same semantic meaning. The user uses the ontological terms to build his queries, but the information that has to be filtered is stored in the columns of the registered databases. Therefore, the ontological terms must be translated into the associated column names before any information can be retrieved to answer the user's query. The mappings required to perform the translation are stored in a database on the server, which keeps track of all of the information required to perform the reverse mapping, including which column names are associated with which ontological terms and the columns that belong to each registered database. When a new query is received, the query is decomposed into individual query terms, which are composed of pairings between ontological terms and filter terms. The ontological term in each query term is translated into a list of associated column names, so that the filter terms can be mapped to the appropriate database column names.

Once the appropriate column names are ascertained, the next step is to figure out which component databases and database tables need to be queried. The information in the knowledgebase is stored in the tables of remote databases, and only a subset of the total information is required to answer a user's query. Therefore, it is important to determine the appropriate subset correctly in order to minimize the amount of work required to answer the query without losing precision. The mappings of column names to database tables and component databases are stored on the server during registration and used to construct a minimal-sized set of databases to query. After the set of component databases that need to be queried is ascertained, the user's query is rewritten into sub-queries that are sent to the appropriate databases.

The user query is rewritten into sub-queries in the SQL format, because of the query model assumed in the system design. The query rewriting process occurs in two steps. The first step involves rewriting the user query into individual sub-queries, where each sub-query involves only one database table. The second step determines whether any of the columns in a sub-query references a column in another sub-query and performs a union of the two queries. The extra step minimizes the amount of network traffic generated in the system by reducing the number of sub-queries sent to the remote databases.

Each sub-query is first constructed from the result of the translation step, where ontological terms are reverse mapped to the appropriate database column names. The column names are grouped together by the databases where they belong. The SELECT clause of the SQL query consists of the unique identifiers of the biological objects stored in the component databases that are pertinent to the query. The unique identifier is only unique to the database queried and is not necessarily globally unique. Once the column names associated with the ontological terms in the query are known, it is possible to determine which database tables need to be queried. The columns to tables mappings were stored during the registration process. The tables containing the columns that need to be queried make up the FROM clause of the SQL query, and each sub-query consists of only one FROM clause. If more than one table is required to answer the query, then multiple sub-queries are created and the sub-queries are grouped together by database. The column name and its filter terms make up the WHERE clause of the sub-queries. The column name must be associated with at least one of the ontological terms in the user query and located in the database tables identified in the FROM clause of the sub-query in order for the column name to be a WHERE clause in that sub-query. If the sub-query contains multiple WHERE clauses, then they are strung together by ORs. OR is used instead of AND in order to maintain a significantly high recall. The steps involved in translation a user query into sub-queries are illustrated in Figure 4.



user query: ontological_term1="filter_term1" ontological_term2="filter_term2" ontological term3="filter_term3"
decomposition
ontological_term1 => "filter_term1"
ontological_term2 => "filter_term2"
ontological term3 => "filter_term3"
translation using mappings between ontological terms and column names
column_name1 => "filter_term1"
column_name2 => "filter_term2"
column_name3 => "filter_term3"
determine table where each column belongs
table1 <= column_name1 => "filter_term1"
table1 <= column_name2 => "filter_term2"
table2 <= column_name3 => "filter_term3"
determine database where each column belongs
database1 <= table1 <= column_name1 => "filter_term1"
database1 <= table1 <= column_name2 => "filter_term2"
database1 <= table2 <= column_name3 => "filter_term3"
rewrite into SQL
optimize by taking UNION

```
SELECT identifier1
   FROM table1
   WHERE column_name1=filter_term1
     OR column_name2=filter_term2;
```

```
SELECT identifier1,identifier2
   FROM table2
   WHERE column_name3=filter_term3;
```

```
SELECT identifier1
   FROM table1
   WHERE column_name1=filter_term1
     OR column_name2=filter_term2
UNION
SELECT identifier1,identifier2
   FROM table2
   WHERE column_name3=filter_term3;
```

Figure 4: Steps involved in translating a user query into sub-queries.

After the sub-queries for each database are constructed, the system checks whether any of the sub-queries can be joined together in order to minimize the amount of network traffic generated by the system. Two queries are joined together if one of the columns in a sub-query is an identifier that references a column in another sub-query, indicating a cross-reference between the two databases. The queries are joined using the UNION clause. A UNION is a sufficient temporary substitute for a JOIN, because only identifiers are currently returned as results, and the union of the identifiers returned from the original sub-queries is similar to taking a JOIN of the original sub-queries with OR clauses.

5.4 Results Aggregation

Results aggregation consist of three steps: transport of results from the component databases to the aggregator, caching and aggregation of the results, and translation of the results into LSIDs. The identifiers returned from each sub-query have to be translated into LSIDs, so that the identifiers are globally unique. The system returns biological identifiers as results from queries to the component databases in order to simplify the design of the system. Several steps in the aggregation process would be more complex if the user were allowed to choose what relations to return. For example, if the user were allowed to choose what relations to return as return values, then result aggregation would be complicated by the fact that not every relation exists in every component database. If the SELECT clauses do not exist in the database even though the WHERE clauses do exist, then the machine would be required to know how to substitute for the missing information. If several relations were selected as return values, then issues associated with displaying the return values in an understandable format must also be resolved. Furthermore, the data model of the return values from different databases could be different, so the aggregation layer would be required to handle that conflict. Transporting non-string values across the network would also be more difficult and could potentially bottleneck the network. Therefore, using object identifiers as return values minimizes network traffic and simplifies the transport, aggregation, and presentation of results.

The aggregation of object identifiers is performed in a database located at the server, because large result sets cannot be stored entirely in memory. Storing the results in a database facilitates the future optimization of the system with threads to handle simultaneous querying of component databases, because unpredictable network traffic prevents the results from different databases from returning to the aggregator at the same time. Therefore, the system needs a repository to store the subset of results returned until a response from all of the databases are received. Storing the results in memory is suboptimal, because memory is an extremely limited resource. Not only is there more space in a database to hold the results, but the database can also act as a cache for recent queries and their results in order to minimize the amount of redundant work performed by the system. The downside of using the database as a repository for the results is the increase in the number of database I/Os.

After the object identifiers from the component databases are aggregated, the identifiers need to be translated into LSIDs in order to make the identifiers globally unique. Ideally, the identifiers returned from the component databases would already be in the LSID format, and the system would query a set of LSID resolvers for information associated with each LSID. However, an extra translation step is currently necessary until LSIDs are adopted by the biological community. Translation is also dependent on the LSID resolvers available to the system, because the identifiers need to be translated into LSIDs that can be interpreted and associated with the appropriate biological objects in the remote data sources. All of the identifiers are currently translated into one LSID format, because there was only one LSID resolver available on the World Wide Web during the implementation of the system. In the future, the system would incorporate more LSID resolvers as they become available in order to make LSID resolving scalable, because it would be difficult for a single resolver to resolve every LSID in the biological knowledgebase.

5.5 Ontology for Database Schema Integration

An important application of biological ontologies is to facilitate database schema integration. Representational heterogeneity in databases regarding how information is structured in table columns makes it difficult for machines to know what data can and cannot be integrated. Data is organized as columns in relational databases, and it is often the case that only the creators of the databases understand the semantic meaning of each column. The problem is further complicated by the fact that column names are not unique in a single database and between different databases. Therefore, transparency is often lost, because it is difficult for a user to query the databases without first understanding how they are structured, and it is impossible for a machine to query the databases without being explicitly told from which columns to retrieve information. Transparency could be restored by mapping the columns to a consistent vocabulary that the user queries to retrieve information.

A biological ontology is a consistent vocabulary of biological terms agreed upon by a group of people. Each term in the ontology is explicitly defined, so that the meaning of each term is clear to everyone who uses the ontology; therefore, ontologies eliminate the semantic ambiguity that hinders communication between different people and machines. The creators of a database schema could add semantic metadata to the schema by mapping each column name to one or more ontological terms that describe the semantic meaning of the column. The mappings would facilitate schema integration, because a machine could use the ontological terms to determine how whether two different columns are semantically identical. If two columns are mapped to the same ontological term, then the columns are considered to be semantically identical and their data could be integrated; otherwise, if the columns are not mapped to the same ontological term, then the columns are semantically different and their data should not be integrated. Therefore, the manual work associated with an individual understanding the definitions of multiple schemas in order to integrate them is no longer required. As long as the creator of mappings between source schema columns and ontologies provides the appropriate mappings, then the system could automatically integrate multiple schemas together based on the schema to ontology mappings.

Likewise, ontologies could also be used in queries to make them semantically meaningful. Different scientists might define a biological term differently depending on the domain in which they work; therefore, naïve pattern matching of keywords is insufficient for retrieving information. The precision of the search can be increased by adding semantic metadata to both the query and the biological information. Semantic metadata is added to biological information by associating ontological terms with database table columns, and metadata is added to a query by incorporating ontological terms into the query. For example, if the user used ontological terms to construct the query, then the machine could use those terms to determine which table columns to query for information. Not only do the ontological terms in the query direct the machine to the location of the information, they also prevent the machine from performing pattern matches on every column in the database.

It is possible that a single static ontology would not have all the terms a user requires to construct his query or the creator of a database requires to map the columns in his schema. An ontology is not a "one size fits all" solution, because it would be extremely difficult to get every biologist to adopt exactly one static ontology. An ontology needs to evolve over time, and multiple ontologies might be required to suit the needs of different people using the system, especially if the people work in different biological domains. Ontology expansion and the support of multiple ontologies is not a problem focused on in the thesis. However, the design of the system is flexible enough to allow the users of the system to choose the ontology they wish to use for integrating database schemas.

5.6 Ontology for Indexing Database Information

A controlled vocabulary is also useful for indexing database information in order to facilitate searching for relevant information. Keywords are useful for minimizing the amount of information that is required to be searched through when determining whether a biological object is relevant to a query or not. However, the naïve pattern matching of keywords provides little value, because one is never sure whether the word used by the user and that in the database have the same meaning. The problem could be solved by indexing the biological objects with ontological terms and allowing the users to construct queries based on those terms. Therefore, keyword searches would become more meaningful, because the searches are performed on semantic keywords with explicit definitions. However, multiple ontologies would be required to serve as indices, because it is impossible for a single ontology to accurately represent all of the information in the biological knowledgebase.

Fortunately, many different biological ontologies are currently being defined, such as GO, in various biological domains and adopted into both data banks and knowledge banks; as a result, biological databases are being automatically indexed by multiple ontologies from the perspective of the system designed in the thesis. Therefore, the system only needs to provide a method for accessing the ontological terms and their association in the databases. As the ontologies become more ubiquitous, one might question whether all of the complexity associated with schema integration is really necessary when one could simply query the ontologies in order to retrieve the desired information from the databases. Although the ontologies are being incorporated into different databases, the ontologies themselves are also stored in databases, so schema integration is still necessary in order to query and integrate multiple ontologies. Therefore, the architecture proposed in the thesis treats an ontology as an individual database that is integrated into the system.

6.0 Architecture

The system consists of a distributed architecture that can be divided into four major components: the user client, the server, the data source, and the LSID resolver. The first section provides a general overview of the system architecture. The second section describes the client interface. The next four sections detail the two main components of the server that handle queries and their results. The following three sections describe the remote data sources and the interface between the data sources and the server. Finally, the last section explains how the LSID resolver fits into the system architecture.

6.1 Overview

The system consist of a distributed instead of a centralized architecture, because the existing biological data repositories that are integrated into the system already exist in a distributed fashion on the World Wide Web. The majority of the biological data repositories are independent databases disconnected from each other. A distributed architecture is also preferred over a centralized architecture, because the distributed architecture provides more flexibility in adding new data sources. The integrated data sources are loosely connected, making the system easier to expand. The architecture also has the advantage of distributing the workload amongst multiple machines, which makes the design more scalable. The distribution of workload also gives the architecture many other advantages that are not easily provided by a centralized architecture.

The two major disadvantages of a centralized architecture are the existence of a single point of failure and the monolithic nature of the system. The components of a centralized architecture are tightly connected, making it hard to expand. Furthermore, the monolithic nature of the centralized system makes it difficult to replicate for backing up the system and reducing the workload on a single system. In contrast, the distributed architecture distributes the workload amongst multiple machines, which remain independent of each other. Therefore, the replication of each component is facilitated by the fact that the components are smaller and not dependent on one another's status. Moreover, the entire system can remain functional even when a component becomes temporarily unavailable, because each component is autonomous. Although the autonomy of the components acting as data sources could generate inconsistencies in information, the inconsistencies are desirable. One goal of integrating data from multiple data repositories is to make the inconsistencies apparent, so that scientists can determine whether more experiments are required to resolve the inconsistent information or that the inconsistencies are natural phenomena. The distributed architecture requires several interfaces to mediate the communication between different components, and to abstract out the details of each component from one another in order to make the overall system more tolerable to changes. Figure 5 shows an overview of the system architecture designed in the thesis.

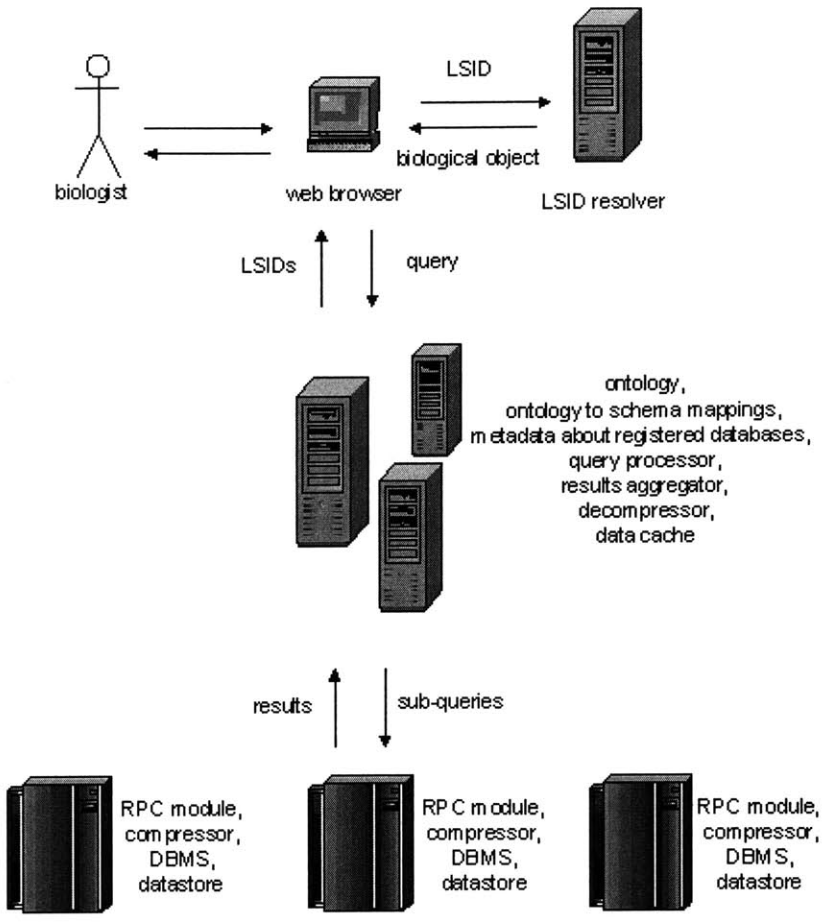


Figure 5: Overview of system architecture.

The two main interfaces in the system are the interface between the user and the server and the interface between the results aggregator and the data sources. The interface between the user and the server determines how queries are constructed and query results are displayed. The interface between the results aggregator and data sources determines how the user queries are decomposed and translated into sub-queries, and how the results are returned from the data sources and aggregated. All of the interfaces are designed to be lightweight in order to promote loose connectivity; however, the looseness creates more work on the system during query processing and results aggregation. One of the main disadvantages of a distributed architecture is the unreliability of the network connection between the different interfaces. The unreliability of the network could easily make the network a bottleneck in the system, so optimizations are made in the design of the architecture to help avoid the bottleneck.

Two optimizations are made in the system architecture to avoid network bottlenecks: caching the queries and their results, and compressing the query results sent from the data source to the results aggregator. The addition of a cache reduces the amount of redundant work performed by the system, and data compression reduces the amount of time required to send a package across the network by reducing the package's size. The downside of the optimizations is the extra work associated with data caching and compression. An extra step is necessary to check whether a user query already exists in the cache. Two extra steps are required to compress and decompress the query results. Furthermore, mechanisms would be required to ensure that information in the cache is not stale, the system makes efficient use of the cache without thrashing, and data is not corrupted during compression and decompression. Fortunately, the extra work could be distributed across different parts of the system.

The flexibility provided in a distributed architecture also facilitates the integration of the architecture proposed by the thesis with other systems, such as the LSID resolver. Therefore, the problem of building a LSID resolver is outsourced, and existing solutions are reused to avoid the reimplementation of the same solution.

6.2 User Interface

The user interface is the user's portal into the system. The user model assumed in the design requires that the interface to construct queries and browse results be simple to use. Other properties that the architecture accounts for are consistency, speaking the user's language, supporting undo, minimizing memory overload, and a minimalist design. The user interface follows platform conventions to ensure that the semantics of the language used and the actions on each page presented to the user are unambiguous. The textual information is presented in the user's language instead of the system's language, and the tasks match real world situations so that the actions required from the user appear natural. Users are allowed to return to previous pages in order to undo errors, and the information presented on each page is kept to a minimum in an effort to minimize the amount of information the user is required to remember between different pages. Finally, the interface follows a minimalist design in order to avoid distracting the user from the task at hand with irrelevant information.

6.3 Schema Mapping

The main component of the interface between the server and the data sources is the mappings between the ontology and source schemas of the databases integrated into the system. The mappings are essential during query processing for reverse mapping the ontological terms to the corresponding database table column names. The process is similar to the reverse mapping in the mediator approach to data integration, where the ontology takes the role of a global schema to which the table columns are mapped. The architecture proposed in the thesis is similar to the LAV (local-as-view) architecture, because the source schemas are a subset of the global schema. Therefore, both share the advantage that it is easy to add new data sources to the system. The mappings are currently stored in a database on the server as pairs. The metadata stored about each registered data repository and ontology are shown in Figure 6.

```
CREATE TABLE Terms (
     code                VARCHAR2(1024) PRIMARY KEY,
     preferred_name      VARCHAR2(1024),
     semantic_type       VARCHAR2(1024),
     definition          VARCHAR2(1024)
);

CREATE INDEX Terms_idx ON Terms (LOWER(semantic_type));

CREATE TABLE DataSource (
     id       INTEGER PRIMARY KEY,
     name     VARCHAR2(1024),
     url      VARCHAR2(1024)
);

CREATE TABLE DataSourceTables (
     id       INTEGER PRIMARY KEY,
     name     VARCHAR2(1024),
     ds_id    INTEGER REFERENCES DataSource(id)
);

CREATE TABLE DataSourceTerms (
     id       INTEGER PRIMARY KEY,
     name     VARCHAR2(1024),
     type     VARCHAR2(1024),
     tb_id    INTEGER REFERENCES DataSourceTables(id),
     refer    INTEGER REFERENCES DataSourceTerms(id)
);

CREATE TABLE TermsMapper (
     thes_term_id       VARCHAR2(1024) REFERENCES Terms(code),
     ds_term_id         INTEGER REFERENCES DataSourceTerms(id),
     tb_id              INTEGER REFERENCES DataSourceTables(id),
     ds_id              INTEGER REFERENCES DataSource(id)
);
```

Figure 6: Database schema of metadata about registered data repositories and ontology.

6.4 Query Processor

The query processor translates the user query into sub-queries that could be answered by the component databases integrated into the system. The process involves query decomposition, reverse mapping of ontological terms to database column names, and query rewriting. The first step decomposes the user query into query terms that consist of an ontological term and its filter terms. The second step uses the schema mappings to translate the ontological terms into column names. The third step rewrites the query terms into SQL queries and performs the appropriate optimizations to minimize the number of queries sent to each component database. After the sub-queries are constructed, each sub-query is sent to the database that can answer the sub-query. The original user query is cached in a database on the server, in case the same query is posed by another user, along with the aggregated results of the query.

6.5 Results Aggregator

The results aggregator aggregates the query results from the component databases, determines the significance of each unique result, translates the results into LSIDs, and returns the LSIDs to the server, which displays the LSIDs to the user. A result is an identifier associated with a biological object stored in one of the component databases. Each result returned from a sub-query is cached in a database on the server with the appropriate user query. After all of the component databases required to answer the query deliver their results, the significance of an identifier is determined by how many times it appears in the aggregate. Identifiers might appear multiple times, because they are cross-referenced by other databases. For example, if a query requests both structural and functional information, and the information is stored in two separate databases that cross-reference each other, then the aggregate result would contain at least two counts of each identifier from both databases related to the query. Once the significances of the identifiers are determined, redundant identifiers are removed and each remaining identifier is are translated into a LSID. The identifiers are translated into LSID formats supported by the LSID resolvers accessible to the system. The LSIDs are returned to the server, which displays the LSIDs according to the specifications of the user interface.

6.6 Data Cache

The data cache is a database on the server that stores the user queries and their results in case a query is posed more than once. A database is used, because it is easy to query and reliable for storing information. The cache reduces the amount of redundant work performed by the system, but also adds some overhead in terms of checking the cache, storing the information in the cache, and maintaining the cache. Mechanisms for maintaining data freshness in the cache and making efficient use of the cache are also required, but these two problems are outside the scope of the thesis. There is currently no size limit to the current cache. The database schema that represents the data cache in the architecture is shown in Figure 7.

```
CREATE SEQUENCE Results_seq START WITH 1 INCREMEMENT BY 1 CACHE 20;

CREATE TABLE Results (
     id         NUMBER,
     result     VARCHAR2(1024)
);

CREATE INDEX Results_idx ON Results(id);

CREATE TABLE ResultsCache (
     results_id     NUMBER,
     query          VARCHAR2(4000)
);

CREATE INDEX ResultsCache_idx ON ResultsCache(results_id);
```

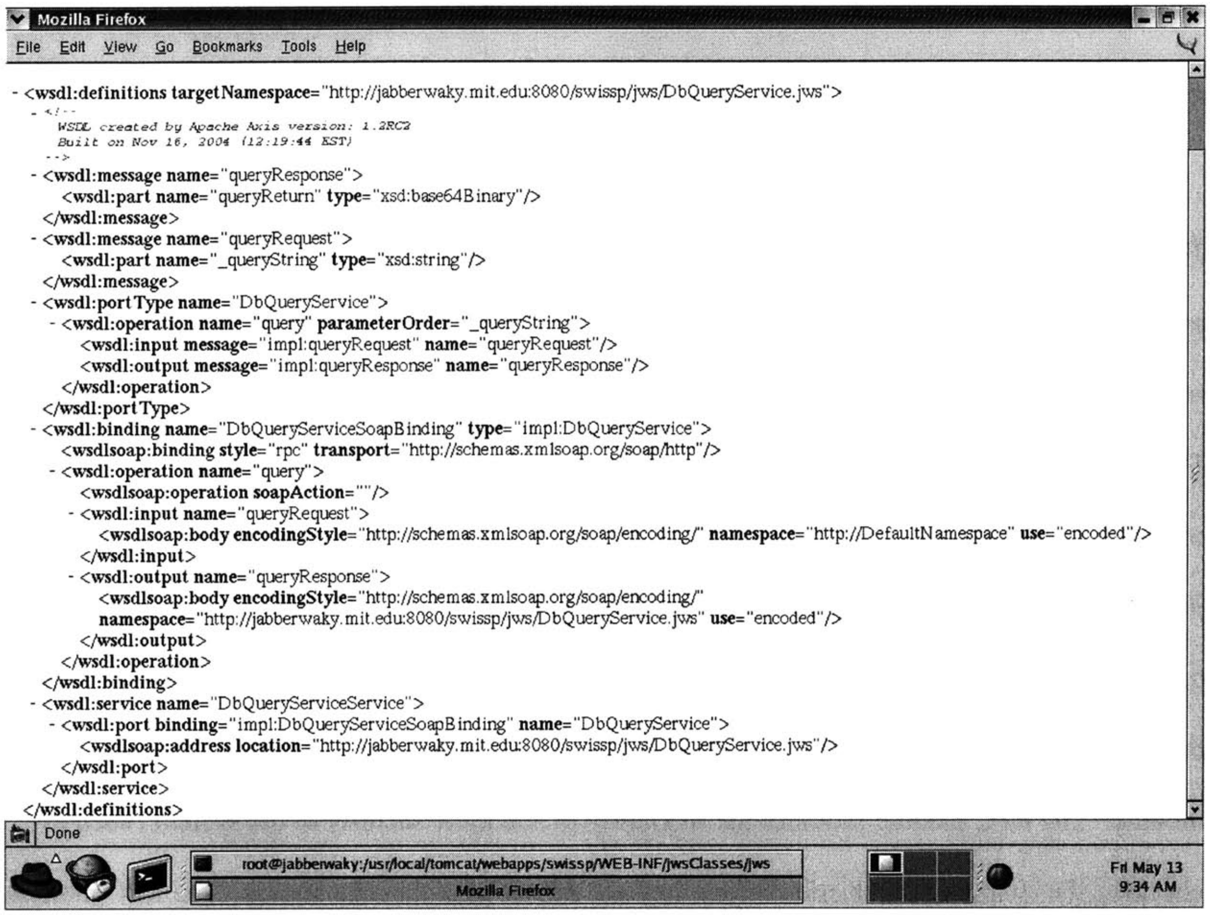
Figure 7: Database schema for data cache.

6.7 Data Storage

Many components of the architecture require data storage, such as the component data repositories that store biological information, the server that stores the mappings between the ontological terms and column names in source schemas, and the components that store the ontology and data cache. The majority of the data stored in the system are in relational DBMSs. However, the SQL commands required by the system are supported by all major DBMSs, and the module sitting above each data source is not specific to any database. Therefore, the system is database independent, and any of the major relational DMBSs can be used as components in the system, including IBM DB2, MySQL, Oracle, and PostgreSQL.

6.8 Remote Procedure Call (RPC)

The communication layer of the architecture that allows machines to communicate with each other in the system is implemented using SOAP RPC. RPC allows two different machines running on different environments to communicate with each other using XML based messages. The RPC module sitting above each data source implements a procedure that is exposed to the outside world, allowing other machines to query the data source through the module. The procedure provides the input and output specifications of the RPC module: it accepts a SQL query and returns a compressed array object that stores the results of the query. Figure 8 shows the Web Service Definition Language (WSDL) file for the web service located at the RPC module. The RPC module provides the flexibility to add more security to the system , such as checking the syntax of the queries before they are delivered to the data source. The RPC module is important for facilitating the distributed design of the architecture and allowing the data sources to remain autonomous.



```
- <wsdl:definitions targetNamespace="http://jabberwaky.mit.edu:8080/swissp/jws/DbQueryService.jws">
  - <!--
      WSDL created by Apache Axis version: 1.2RC2
      Built on Nov 16, 2004 (12:19:44 EST)
    -->
  - <wsdl:message name="queryResponse">
      <wsdl:part name="queryReturn" type="xsd:base64Binary"/>
    </wsdl:message>
  - <wsdl:message name="queryRequest">
      <wsdl:part name="_queryString" type="xsd:string"/>
    </wsdl:message>
  - <wsdl:portType name="DbQueryService">
    - <wsdl:operation name="query" parameterOrder="_queryString">
        <wsdl:input message="impl:queryRequest" name="queryRequest"/>
        <wsdl:output message="impl:queryResponse" name="queryResponse"/>
      </wsdl:operation>
    </wsdl:portType>
  - <wsdl:binding name="DbQueryServiceSoapBinding" type="impl:DbQueryService">
      <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    - <wsdl:operation name="query">
        <wsdlsoap:operation soapAction=""/>
      - <wsdl:input name="queryRequest">
          <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="http://DefaultNamespace" use="encoded"/>
        </wsdl:input>
      - <wsdl:output name="queryResponse">
          <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="http://jabberwaky.mit.edu:8080/swissp/jws/DbQueryService.jws" use="encoded"/>
        </wsdl:output>
      </wsdl:operation>
    </wsdl:binding>
  - <wsdl:service name="DbQueryServiceService">
    - <wsdl:port binding="impl:DbQueryServiceSoapBinding" name="DbQueryService">
        <wsdlsoap:address location="http://jabberwaky.mit.edu:8080/swissp/jws/DbQueryService.jws"/>
      </wsdl:port>
    </wsdl:service>
  </wsdl:definitions>
```

Mozilla Firefox
File Edit View Go Bookmarks Tools Help
Done
root@jabberwaky:/usr/local/tomcat/webapps/swissp/WEB-INF/jwsClasses/jws
Mozilla Firefox
Fri May 13
9:34 AM

Figure 8: WSDL file for the web service located at the RPC module.

6.9 Data Compression

Data sent across the network is compressed in order to minimize the amount of network traffic in the system and avoid network bottlenecks. The network is expected to be one of the bottlenecks in the architecture, because all of the communication between different machines in the system is performed through the network. The compressor intercepts the query results returned from the databases and uses the JAVA compression API to compress the results. The decompressor intercepts the results from the component databases and decompresses the results before they are processed by the results aggregator. The compression and decompression steps require extra resources and time; however, it is expected that the decrease in the amount of time each package spends on the network will benefit the system more.

6.10 LSID Resolver

The LSID resolver translates a LSID into the associated biological object and returns information about the object to the user. Although a customized LSID resolver could be constructed for the system, it makes more sense to use existing resolvers on the World Wide Web. It makes sense to outsource the problem, because designing LSID resolvers is not the focus of the thesis and several groups, such as the BioPathways Consortium (http://lsid.biopathways.org/resolver/) and BioMoby (http://mobycentral.cbr.nrc.ca/cgi-bin/LSID_Resolver.pl), are already building LSID resolvers and making them available on the Internet. Therefore, it makes more sense to take advantage of the distributed architecture and reuse an existing solution than to spend time reimplementing the exact same solution.

7.0 Implementation

The following sections detail how the design and architecture proposed in the previous sections are implemented in the system. The implementation focuses on answering questions similar to the ones posed in the Introduction concerning proteins due to limitations on the resources available. The first two sections describe the hardware and software used to construct the system. The next three sections describe how the biological data repositories, ontologies, and unique identifiers are implemented to support the system designed. The last two sections discuss the simplifications made during the implementation of the system and the limitations of the implementation.

7.1 Hardware

Two machines were available to implement the architecture proposed for the system. Therefore, one machine (Machine 1) is dedicated to simulate the remote biological data repositories with the RPC modules, while the other machine (Machine 2) performs all of the other tasks necessary to implement the design, including the user interface, the server, and the results aggregator. Machine 1 has a 700 MHz Pentium III processor and 512 Mb of RAM. Machine 2 has a 1.6 GHz Pentium IV processor and 512 Mb of RAM. More processor power is provided to Machine 2, because it performs a wider variety of tasks, all of which require a significant amount of processing power. The two machines are physically separated and connected by the M.I.T. network, which has speeds that range from 10 Mb/sec to 100 Mb/sec.

7.2 Software

The software environments of the two machines are extremely similar to each other. Both machines run Red Hat Linux 9 as the operating system and use Java as the programming language for applications development. The development process is facilitated using Apache Ant for organizing and building the application. Java Servlets and JavaBeans are used to construct the business layer with Apache Jakarta Tomcat as the servlet container. The front end is built using JavaServer Pages with JavaServer Pages Standard Tag Library (JSTL), JavaScript, and HTML. RPC using SOAP is implemented with Apache Axis. SOAP is preferred over Java RMI, because a lightweight and expandable architecture is desired. Although Java RMI facilitates the transport of Java objects, it is a heavyweight solution. All of the data storage components are built using Oracle Database 10g. The module for uploading files to the server is built using the servlet support classes provided at the Servlets.com website (http://servlets.com). All of the software used to implement the system proposed by the thesis is free for academic use and can be found on the World Wide Web. Figure 9 lists all of the software required to build the system and the use of each component.

Software	**Use**
Apache Jakarta Tomcat 5.5.4	Servlet container for Java Servlets, JavaServer Pages, and JavaBeans
Apache Axis 1.2RC2	Implementation of SOAP (Simple Object Access Protocol)
Apache Ant 1.6.2	Java based build tool
COS package from Servlets.com	Servlet support classes for uploading files
Oracle Database 10g	Database Management Server
J2SE 5.0	Java environment for applications development
JSTL 1.2	Tag library for writing JavaServer Pages
Red Hat Linux 9	Operating System

Figure 9: All of the software required to implement the system and their uses.

7.3 Biological Data Repositories

The data repositories are chosen in order to demonstrate how the architecture designed could be used to integrate data from heterogeneous databases and answer questions similar to the ones posed in the Introduction. Only two data repositories are implemented in the system due to a lack of time: Swiss-Prot and Prosite. Swiss-Prot is a highly annotated database that contains protein sequence information, and Prosite is a database of protein families and domains. Both databases cross-reference each other through their unique accession numbers. Therefore, the system can answer queries related to protein sequence, structure, and function by integrating information from both data repositories. The data repositories communicate with the results aggregator through the RPC module, which is implemented in Java. The Oracle DBMS that controls access to the data repositories, such as Swiss-Prot and Prosite, is connected to the RPC module using a JDBC connector.

7.4 Biological Ontologies

Two biological ontologies are used in the system to demonstrate how ontologies could facilitate schema mapping for integrating heterogeneous databases and indexing biological information to facilitate searching for information. The system uses the NCI Thesaurus as the ontology for performing schema mapping, because it contains a broader range of biological terms than most other existing ontologies. A broad range of biological terms is desirable for integrating databases from multiple biological domains. The system is flexible enough to substitute a different ontology for the NCI Thesaurus; however, the system only supports using one ontology at a time to perform schema mapping.

The Gene Ontology (GO) is used for indexing biological information in biological data repositories, because the ontology is already widely incorporated into several biological databases, including Swiss-Prot and Prosite. The information is indexed, because each biological object in Swiss-Prot and Prosite is associated with zero or more GO terms that are related to the object. However, the system is designed to incorporate any number of ontologies that index biological data repositories, as long as the ontology could be stored in a relational database and is cross-referenced in the biological data repositories integrated into the system. GO is the only ontology incorporated into the system to demonstrate how the ontology could be used to facilitate searching for information in both Swiss-Prot and Prosite.

7.5 Biological Unique Identifiers

Although each of the biological data sources, including Swiss-Prot, Prosite, and GO, have its own method for uniquely identifying the biological objects stored in the data source, the identifiers are not globally unique. Consequently, it is possible for two or more biological identifiers to clash, meaning that different biological objects from different data sources have the same biological indentifer. The clashing makes data access and sharing extremely difficult. The system designed in the thesis prevents the identifiers it uses from clashing by translating them into LSIDs.

The use of LSIDs requires LSID resolvers to return biological information associated with each LSID to the user. The system is integrated with the BioPathways Consortium LSID resolver client, which is part of the BioPathways Consortium Life Science Identifier project. Therefore, the LSIDs in the system are translated into the LSID format defined by the BioPathways Consortium. However, a biological identifier used in the system could be translated into any other LSID format, as long as the system knows which LSID resolver is able to resolve that specific format. The architectural design does not limit the number of the LSID resolvers that could be integrated into the system.

7.6 Simplifications

Several simplifications are made to the system during the implementation process due to limitations in the amount of time, hardware, and existing solutions that could be incorporated as system components available. Some of the simplifications required redesigning the architecture of the system, while other simplifications are temporary and could be easily removed later when more resources become available. The major simplification made is to replace ontologies written in the Web Ontology Language OWL (OWL) [47] with ontologies stored in relational databases. Other simplifications include returning only the first 200 results from each of the system's data repositories to the results aggregator, and using only a subset of the information from the NCI Thesaurus, Swiss-Prot, and Prosite in the system's data repositories.

One of the original goals of the thesis was to design a Semantic Web application that used ontologies written in OWL to integrate source schemas from heterogeneous databases. OWL is a component of the Semantic Web project [48] from the World Wide Web Consortium (W3C) that represents ontologies as Resource Description Framework (RDF) [49] graphs. RDF is another component of the Semantic Web project that is used to provide metadata about resources on the World Wide Web. The term "resource" is used in its broadest sense and encompasses a single word to an entire database. The metadata facilitate semantic reasoning about the information by machines. Therefore, ontologies represented in OWL contain semantic metadata that could be useful for facilitating information reasoning by machines.

Another advantage of using OWL to represent ontologies is that the syntax of RDF is similar to the syntax of XML, but RDF is more flexible than XML in how information is represented. Therefore, OWL provides a flexible syntax for representing different ontologies, which could be exploited in the system architecture to represent source schemas, since a database schema is extremely similar to an ontology. For example, a relational database schema is hierarchical (database → table → column), and every component at each level of the hierarchy has semantic meaning. OWL could also be useful for making the implicit information in a source schema, which is usually only known to the person who creates the schema, explicit as metadata. The metadata could be used during schema integration to determine how two schemas that use different RDF tags should be integrated with the help of an OWL parser and reasoner. If the schemas used the same RDF tags, then the OWL parser alone is sufficient for integration.

Likewise, OWL could also be used to represent the mappings between ontological terms and database columns, because the OWL syntax facilitates the representation of semantic relationships, such as synonyms and antonyms. If an ontological term and a database column share the same semantic meaning, then the two objects could be stored as synonyms in the OWL document. The mapping could be used during query processing with the help of an OWL reasoner. Therefore, the design stored the ontology used for querying, and the mappings between ontological terms and database columns as persistent OWL documents. By representing the database schemas and mappings in OWL, it could become easier to exchange schemas and integrate them into any system, as long as a mechanism exists for parsing and reasoning about the OWL document.

Unfortunately, the architecture implemented could not support the use of OWL due to limitations in the technologies available to parse and reason about persistent OWL ontologies. The main Java application currently available for handling OWL ontologies is the Jena Semantic Web Framwork (Jena) [50] from Hewlett Packard. The version of Jena used during the implementation phase of the system was Jena 2.2. Jena was chosen, because it has the most support and number of developers working on the project compared to other Java projects that deal with OWL ontologies. Jena could not be used for two major reasons: it does not scale and is unreliable.

Initial experiments with loading and querying small ontologies that were on the order of kilobytes in size with Jena were successful. However, when the 57 Mb NCI Thesaurus was loaded into Jena as a persistent ontology, the loading process required on the order of tens of minutes, and querying the persistent ontology took on the order of minutes. Jena becomes extremely slow when the ontology is large, which is a major problem when it is a component of a web application. Users today are accustomed to receiving results on the order of milliseconds from any application; even a couple of seconds is considered to be too slow. Therefore, Jena becomes unusable if it takes one hundred times longer to perform a single task. Furthermore, several software bugs were discovered in Jena while working with the persistent model, and the bugs had to be fixed in order to implement the architecture designed in the thesis. Debugging the Jena software quickly became time consuming, and Jena was no longer an independent black box component that could be integrated into the system architecture.

Consequently, Jena was removed from the architecture designed, because it causes the application to becomc too slow, and the bugs in the software impeded the implementation of the architecture proposed by the thesis. Some time was spent looking at other Java applications for handling OWL ontologies, but all of them were unique with their own Application Program Interfaces (APIs) and quirks. All of the applications were also in the developmental phase. Writing a custom OWL parser and reasoner was another possible solution, but it would have taken too long and was outside the scope of the thesis. Therefore, it was decided not to attempt at integrating another Java application for handling OWL documents; instead, a simplification was made to the architecture of the system and implemented as part of the thesis.

The simplification consists of replacing the persistent OWL ontology with an ontology stored in a relational database and registering databases with the original SQL statements used to create the database tables. Loading the NCI Thesaurus now takes on the order of minutes and querying the ontology takes on the order of seconds. The redesigned architecture only stores information about the name of the ontological term and its definition. The architecture does not store any information about how one ontological tcrm is rclated to another – hierarchical information.

Although the new representation of the ontology is not as powerful as the OWL representation, it is sufficient for demonstrating how an ontology could be used to facilitate the integration of information from multiple heterogeneous databases. The hierarchical information could be added to the architecture in the future by expanding the relational database schema used to represent the ontology. The existing architecture should support the expansion without incurring any significant performance penalty.

The other simplifications made to the architecture during the implementation process are less severe and could be easily removed in the future when more hardware becomes available to implement the system. The first simplification involves using only a subset of the NCI Thesaurus in the user interface to construct queries. The NCI Thesaurus contains approximately 20,000 terms, and the terms are cached at the server the first time they are loaded, because it is not expected that the ontology would change very often. However, the client's web browser must reload the terms every time the query construction page is displayed, and displaying 20,000 terms could easily overwhelm any browser and take a significant time to load. Therefore, only the subset of the ontological terms in the NCI Thesaurus that pertains to the queries described in the Introduction is used, which decreases the workload on the web browsers and still proves the point of the thesis. One solution to the problem regarding the size of the NCI Thesaurus is discussed in the Discussions section under Query Processing.

Similarly, only a subset of the information from both the Swiss-Prot and Prosite data repositories are used during the implementation of the system due to a lack of hardware resources. The Swiss-Prot, Prosite, and GO data reside on a single machine in different databases, and the three databases together become extremely resource intensive. Therefore, just enough information is loaded into the data repositories in the system from Swiss-Prot and Prosite to answer queries related to the ones posed in the Introduction. Although the range of queries that the system could answer is small, the point of the thesis is still made with the amount of information available in the system. This simplification would not normally be required, because the data repositories already exist in different machines distributed across the World Wide Web.

Finally, the number of results returned from each component database is limited to a maximum of 200 due to a lack of memory for aggregating results. The maximum number of results is chosen arbitrarily; 200 is an extremely low estimate of what the system could handle. The results aggregator must decompress all of the packages returned from the component databases, store the results in a database on the server, and determine the significance of each result; consequently, a significant amount of CPU power and memory is required. As the number of results increases, some of the processes will begin to use the hard disk as temporary storage when the memory becomes full, slowing down the entire application. Therefore, the number of results returned from each database is temporarily capped at 200, because not all of the memory is available to the results aggregator. Some of the memory is constantly used by the Servlet Container and Oracle. After the Oracle DBMS and Servlet Container are initialized, only 6 Mb out of the 512 Mb of free memory is available, and the amount of free memory available is the same on both machines. The majority of the memory is used up by the Oracle DBMS. It is possible that the combination of a better query processor and results aggregator could avoid the need for this simplification. However, like all of the other simplifications in the architecture made due to a lack of hardware resources, the cap could easily be removed so that all of the results are aggregated.

7.7 Limitations

The architecture proposed is reminiscent of a large DBMS that queries databases instead of tables for information. The implementation of the architecture is limited by two of the most difficult components to build in the system: the query processor and results aggregator. Both components require high levels of expertise in implementing DBMSs to build and optimize correctly. The query processor and results aggregator in the architecture are separate from their corresponding components in the relational DBMS. The components in the architecture span multiple databases, whereas those in the relational DBMS only deal with one database. Therefore, there exist slight differences in how queries are constructed and how results are aggregated. Nevertheless, many of the concepts behind building the DBMS components carry over to building the components in the architecture proposed by the thesis, because they share extremely similar functionalities.

Similar to the query processor in a relational DBMS, the query processor in the architecture proposed in the thesis must optimize the user query for retrieving information in the most efficient manner. The different steps involved in query processing include query construction, decomposition, translation, and rewriting. Query processing is complicated by the fact that the system is similar to a LAV (local-as-view) architecture. The difficulty in implementing a query processor led to many of the limitations associated with the query processor built for the system.

First, the query constructor does not support Boolean operators, such as NOT, AND, and OR. Second, query translation is limited to translating the user query into SQL sub-queries that only involve SELECT and UNION. Third, query rewriting can only optimize the sub-queries when a column selected in one sub-query references another selected column in a different table in another sub-query by taking the UNION of the two sub-queries. The system cannot rewrite sub-queries into JOIN statements. Another limitation that influences query processing is the lack of statistics regarding access to different databases.

Relational DBMSs usually store statistics about how tables in the database are accessed in order to optimize how queries are decomposed. The architecture implemented does not store usage statistics about the component databases, making it difficult for the query planner to decide which database to query for information if redundant information is stored in multiple databases. Currently, every component database related to a query is accessed, which could lead to redundant work being performed. Unlike the single relational DBMS, the architecture in the thesis consists of an extra network layer that could become a bottleneck, so it is important to minimize the amount of redundant work performed by the system. Although the query processor implemented is sufficient for demonstrating the goal of the thesis, the limitations described above are worth taking note of when implementing a system that is intended for public use. The final implementation of the query processor used in the system is shown in Appendix I.

Likewise, the results aggregator is another component that requires expert knowledge to implement correctly. Results aggregation involves making sense of the information returned from the component databases and performing horizontal integration, vertical integration, or both. Most existing integration platforms only perform horizontal integration. The architecture implemented also performs horizontal integration and has the capability to perform vertical integration using the mapping information between the ontological terms and database columns. However, vertical integration is not implemented, because the component databases are designed to only return biological object identifiers as query results and the Swiss-Prot accession numbers do not overlap with the Prosite accession numbers, and these accession numbers do not overlap with the GO identifiers.

The architecture only returns biological object identifiers from the component databases, because they can be easily translated into LSIDs with existing LSID resolvers, are easier to transport across the network, and function like URLs. Returning identifiers is sufficient for the thesis, but it might be desirable to return more information to the user from the databases. However, methods of dealing with the problems associated with returning more information described earlier would be required. Similar to query processing, results aggregation is a complicated task that involves several tradeoffs between different designs. Although the design implemented in the system is sufficient for the purposes of the thesis, the limitations described are still worth some thought. The final implementation of the results aggregator used in the system is shown in Appendix II.

8.0 Discussion

The following sections discuss the difficulties encountered in the areas of schema integration, query processing, and results aggregation during the design and implementation of the proposed architecture. The last section investigates how the system could be incorporated into the Semantic Web.

8.1 Database Schema Integration

The design proposed in the thesis uses an ontology to make the heterogeneity in databases at the schema level transparent to the user. The approach makes use of the correspondences between different database columns on the language level to facilitate the integration of heterogeneous databases. Although the thesis successfully demonstrates how ontologies could be used to integrate information from multiple databases, several issues related to the proposed architecture are worth mentioning.

The first issue is the problem of vertical integration across multiple databases. The information in the component databases could be complementary or overlapping, and the same data could have different representations at both the schema and data levels in different data sources. Therefore, a mechanism is necessary to resolve the differences at both levels in order to successfully integrate information from different data sources. Most integration systems perform horizontal integration, but no system has demonstrated the ability to perform vertical integration successfully.

Vertical integration is difficult, because it requires that the machine automatically determine what information is semantically the same and how to handle overlapping data during query processing and results aggregation. The mappings between the ontology and database columns used in the architecture facilitate the process of vertical integration at the schema level, because the mappings indicate which columns are semantically identical. However, the mappings are not sufficient for performing vertical integration at the data level. Using ontologies to index the data in the databases adds metadata to the information, which facilitates vertical integration at the data level by providing a common data model through which a machine can compare two pieces of information that might have different data models. Ontologies are successfully implemented in the thesis to solve the problem at the schema level, but all of the data in the data repositories are strings. Therefore, it is unclear whether the ontological indices are sufficient for facilitating vertical integration at the data level.

Ontologies are shown to facilitate data integration from heterogeneous databases, but the integration process is still not automatic. The system requires a user to create the mappings between the ontological terms and database columns, and the mappings might change depending on how the user interprets the source schemas. The mappings play an essential role during query processing and affect the results returned from the component databases. Therefore, it might be necessary to support multiple ontologies and sets of mappings in order to allow users to customize their own view of the biological information. The other possibility is to only allow the database administrator for each component database to create the mappings between the ontology and component database columns, so that the system consists of only one set of mappings. The user could choose to use the system depending on whether he agrees with the mappings created by the administrators.

8.2 Query Processing

Query processing is extremely complicated and resource intensive, making the query processor difficult to implement. As a result, a couple of simplifications are made during the implementation of the architecture for the thesis. Although one general solution could be to wait for more powerful hardware to become available to the system before removing the simplifications, redesigning certain parts of the architecture might also resolve some of the problems without requiring better hardware.

One simplification that is dependent on a resource external to the system is displaying only a subset of the NCI Thesaurus for query construction. In this case, the limiting resources are on the client side, because it is the client's web browser that must render 20,000 ontological terms. Instead of trying to display all of the ontological terms, of which the user only requires a subset at any given time, an alternate design might be to incorporate a search interface into the query construction interface. The user could provide keywords about ontological terms that he is interested in, and only the terms related to those keywords are displayed. Whether an ontological term is related or not could be determined by the metadata associated with each term. The tradeoff in the design is that the interface is less resource intensive on the client browser, but more resource intensive on the server. However, the resource limitation on the server side is probably preferable to the limitation on the client side, because the system could deal with limitations on the server side more readily than those on the client side.

One alternative design to simplify the query processor is to replicate the tables or columns necessary for answering the query in a local database and perform the user query over the local database. Although the alternative design facilitates the use of JOINs and any type of SQL command, there are several problems associated with the design. First, moving tables or entire columns of data could easily bottleneck the network, especially when there are large binary data stored in the tables or columns being transported. Second, the design might move a lot of data that ends up being only used once, which would make the system extremely efficient. Third, the architecture runs into the risk of becoming a data warehouse and requiring large quantities of disk space on the server, because the system would try to cache as much information as possible in order to minimize the average cost of replicating entire tables worth of data.

8.3 Results Aggregation

The absence of globally unique biological identifiers creates the possibility for two identifiers from different databases to clash. Although LSIDs are incorporated into the architecture, the possibility for clashing is not completely removed. Biological identifiers are translated into LSIDs after the identifiers returned from the component databases are aggregated. Therefore, it is possible for two identifiers to clash during aggregation without being noticed by the system, because the identifiers are not globally unique until they are translated into LSIDs. Clashing is avoided in the architecture implemented, because the identifiers associated with the biological repositories incorporated into the system are globally unique with respect to the system. However, better system designs might be required to ensure that identifiers never clash. Two solutions are to either move the LSID translator directly before the results aggregator or directly after the results are returned from the component DBMS. The problem associated with unique identifiers would disappear once LSIDs are adopted by the biological community, so that all biological identifiers exist in the LSID format and no translation between different formats is required.

Results aggregation is also complicated by a limitation on the amount of shared resources, such as memory, so temporary simplifications are made during the implementation of the architecture. The cap placed on the maximum number of results returned from each component database in response to a sub-query is implemented, because of limitations in the amount of resources available to process all of the results returned. It is observed that the amount of free memory not used by the Oracle DBMS and Tomcat Servlet Container is approximately 6 Mb. Although a better query processor and results aggregator might temporarily avoid the need for more resources, the problem still remains unresolved. It is likely that some queries will generate large quantities of results independent of how the components in the architecture are implemented. Therefore, the need for more resources appears inevitable.

8.4 Semantic Web Application

The architecture implemented facilitates the automation of information integration from autonomous, heterogeneous biological databases using ontologies to describe the source schemas and information stored in the data repositories. The system could be incorporated into the Semantic Web if methods of reusing some of the components implemented are available. The original design attempted to facilitate the reuse of the source schemas and ontology using OWL.

The advantage of representing an ontology in OWL over representing an ontology in a database is that the OWL syntax makes it easy to describe ontologies. Although OWL is not implemented as part of the final design due to limitations in existing technologies to manage OWL documents, this section discusses how the architecture designed in the thesis could be integrated into the Semantic Web if the original design were implemented.

The original architecture uses OWL for exchanging source schemas during the registration process and for storing the ontology mapped to columns in databases integrated into the system. There are several advantages to representing the source schemas in OWL. First, all the source schemas would be represented in a common format, which facilitates the exchange of schemas and the registration process. If all of the schemas are represented in a standard format, then only one parser would be required to extract the necessary information required for registering a new database with the system. The ability to add metadata to the OWL document is useful for allowing people other than the database administrators and machines to understand what information is represented in the database. The metadata could potentially be used for adding AI to the system in order to automate the process of mapping ontological terms to database columns. One disadvantage of representing source schemas in OWL is that the database administrators must make the effort to translate the database schema into the OWL format.

Likewise, the main advantage of storing the ontology in the OWL format is the ease of exchanging the ontology and adding metadata to the ontology. If the system supported multiple ontologies, where each ontology is customized to the user's view of the biological knowledgebase, then exchanging ontologies becomes important. It is likely that people do share common views on certain aspects of the knowledgebase; therefore, the ability to exchange parts of an ontology would prevent users from having to recreate those parts. Furthermore, the metadata associated with the ontological terms facilitate the development of AI to make inferences about the information in the ontology, which could potentially be used to facilitate the process of mapping ontological terms to database columns.

9.0 Future Work

The thesis successfully demonstrates how ontologies could facilitate integrating data from multiple heterogeneous relational databases. However, several improvements were realized during the implementation of the architecture designed. The improvements can be divided into two categories: ontologies and architecture.

The improvements related to ontologies include incorporating OWL, supporting multiple ontologies, ontology expansion, and making use of the hierarchical information in ontologies. The advantages of representing the ontologies in OWL and the utility of supporting multiple ontologies are described in earlier sections. Ontology expansion refers to the need to support changes to the ontology mapped to database columns, such as the addition of new ontological terms, as new databases are integrated into the system. The hierarchical information embedded in an ontology is useful to take advantage of, because a significant amount of biological information is also hierarchical. For example, a molecule could be a "protein" and a member of a particular "protein family". Both the terms "protein" and "protein family" could be ontological terms. Therefore, the machine might use the hierarchical information to make the assumption that a member of a “protein family” also has the properties of a “protein” during query processing, and use that inference to return more meaningful results.

Many of the architectural improvements that could be applied to the system are mentioned in earlier sections. The improvements include a less resource intensive way to display large thesauri used to construct queries, support for multiple LSID resolvers, a better query processor, a better result aggregator, and support for vertical integration. One improvement that was not mentioned is asynchronous querying of remote databases using threads. A user query is usually decomposed into several sub-queries that involve multiple databases. The current implementation accesses the databases in series, but a more efficient architecture might access the databases in parallel using threads. Another improvement is the use of smarter caches for maintaining data freshness and storing sub-queries instead of entire queries. Finally, a possible improvement for optimizing query processing is to integrate the DiscoveryLink system into the architecture. DiscoveryLink is a middleware application without a front-end, and the application focuses primarily on query optimization in platforms that involve multiple heterogeneous data repositories. Since the system implemented for the thesis requires a better query processor, integrating DiscoveryLink with the current architecture might solve many of the problems related to query processing.

10.0 Conclusion

The rapid emergence of autonomous biological data sources on the World Wide Web as a result of the large quantities of data generated by high-throughput experiments created a need to integrate information from multiple databases. Integration is essential for data mining; however, current methods for integration require scientists to perform a significant amount of manual work that is both tedious and time consuming. Much time is also wasted learning different user interfaces and resolving conflicts in data representation across different data sources. The thesis proposes a flexible architecture that uses ontologies to facilitate the automation of data integration from multiple heterogeneous databases.

The thesis is a proof of concept that demonstrates how ontologies could facilitate integrating information from heterogeneous databases. Biology is an appropriate use case, because there exists a strong need in the field to integrate information from multiple data sources, and that need remains unmet by existing applications. The thesis tackles the problem on multiple levels: user interface, query processor, results aggregator, communication layer, and data layer. However, the innovation of the thesis is in incorporating ontologies into the system to facilitate query processing by resolving semantic conflicts during schema integration and searching for relevant information. The system also tackles the problem of global unique identifiers for facilitating the exchange and access of information by promoting the use of LSIDs. The system implemented successfully demonstrates that ontologies could facilitate data integration whilmeeting most of the design goals, but several improvements are required before the system is ready for public use, as described in the Future Works section. As more biological data are generated everyday, an architecture is necessary to facilitate the exchange, access, and mining of information in order to make the large quantities of data useful and less overwhelming. The architecture proposed in the thesis is a first step towards a distributed solution to the problem.

11.0 References

1. Brazma, A., et al., *ArrayExpress--a public repository for microarray gene expression data at the EBI.* Nucleic Acids Res, 2003. **31**(1): p. 68-71.
2. Riffle, M., L. Malmstrom, and T.N. Davis, *The Yeast Resource Center Public Data Repository.* Nucleic Acids Res, 2005. **33 Database Issue**: p. D378-82.
3. Benson, D.A., et al., *GenBank: update.* Nucleic Acids Res, 2004. **32**(Database issue): p. D23-6.
4. Berman, H.M., et al., *The Protein Data Bank.* Nucleic Acids Res, 2000. **28**(1): p. 235-42.
5. Bairoch, A. and R. Apweiler, *The SWISS-PROT protein sequence database and its supplement TrEMBL in 2000.* Nucleic Acids Res, 2000. **28**(1): p. 45-8.
6. Altschul, S.F., et al., *Basic local alignment search tool.* J Mol Biol, 1990. **215**(3): p. 403-10.
7. Sigrist, C.J., et al., *PROSITE: a documented database using patterns and profiles as motif descriptors.* Brief Bioinform, 2002. **3**(3): p. 265-74.
8. Hernandez, T. and S. Kambhampati, *Integration of Biological Sources: Current Systems and Challenges Ahead.* SIGMOD Record, 2004. **33**.
9. BACIIS, http://mensa.sl.iupui.edu:8060/baciis/index.html.
10. Davidson, S.B., et al., *BioKleisli: A Digital Library for Biomedical Researchers.* International Journal on Digital Libraries, 1996.
11. BioNavigator, http://www.bionavigator.com/.
12. Haas, L.M., et al., *DiscoveryLink: A system for integrated access to life science data sources.* IBM Systems Journal, 2001. **40**(2): p. 489-511.
13. Pubmed, E., http://www.ncbi.nlm.nih.gov/entrez/query.fcgi.

14. GUS, http://www.gusdb.org/.

15. Gupta, A., B. Ludascher, and M.E. Martone, *Knowledge-Based Integration of Neuroscience Data Sources.* 12th International Conference on Scientific and Statistical Database Management, 2000(July 26 - 28, 2000): p. 39-53.

16. SRS, http://srs.embl-heidelberg.de:8000/srs5/.

17. TAMBIS, http://imgproj.cs.man.ac.uk/tambis/.

18. Brazma, A., et al., *Minimum information about a microarray experiment (MIAME)-toward standards for microarray data.* Nat Genet, 2001. **29**(4): p. 365-71.

19. Harris, M.A., et al., *The Gene Ontology (GO) database and informatics resource.* Nucleic Acids Res, 2004. **32**(Database issue): p. D258-61.

20. Stoeckert, C.J. and H. Parkinson, *The MGED ontology: a framework for describing functional genomics experiments.* Comparative and Functional Genomics, 2003. **4**(1): p. 127-132.

21. Widom, J., *Integrating Heterogeneous Databases: Lazy or Eager?* ACM Computing Surveys, 1996. **28A**(4).

22. Sujansky, W., *Heterogeneous database integration in biomedicine.* J Biomed Inform, 2001. **34**(4): p. 285-98.

23. Mulder, N.J., et al., *InterPro: an integrated documentation resource for protein families, domains and functional sites.* Brief Bioinform, 2002. **3**(3): p. 225-35.

24. Karp, P.D., et al., *BioWarehouse – Database Integration for Bioinformatics.* 2004.

25. DataJoiner, http://www-306.ibm.com/software/data/integration/db2ii/supportdatajoiner.html.

26. Oracle, http://www.oracle.com/database/index.html.

27. DB2, http://www-306.ibm.com/software/data/db2/udb/.

28. Stodder, D., *Virtualize This!* Intelligent Enterprise, 2003.

29. McCormick, P.J., *Designing Object-Oriented Interfaces for Medical Data Repositories*, in *Department of Electrical Engineering and Computer Science*. 1999, Massachusetts Institute of Technology: Cambridge.

30. Fu, B., *Design of a Genetics Database for Gene Chips and the Human Genome Database*, in *Department of Electrical Engineering and Computer Science*. 2001, Massachusetts Institute of Technology: Cambridge. p. 80.

31. Wilkinson, M.D. and M. Links, *BioMOBY: an open source biological web services proposal.* Brief Bioinform, 2002. **3**(4): p. 331-41.

32. Uschold, M., et al., *The Enterprise Ontology.* The Knowledge Engineering Review, 1998. **13**(1): p. 31-89.

33. Stevens, R., C.A. Goble, and S. Bechhofer, *Ontology-based Knowledge Representation for Bioinformatics.* Brief Bioinform, 2000. **1**(4): p. 398-414.

34. OBO, http://obo.sourceforge.net/.

35. NCIt, http://nciterms.nci.nih.gov/NCIBrowser/Dictionary.do.

36. Schulze-Kremer, S., *Ontologies for Molecular Biology.* Proceedings of the Third Pacific Symposium on Biocomputing, 1998: p. 693-704.

37. Chen, R.O., R. Felciano, and R.B. Altman, *RiboWeb: Linking Structural Computations to a Knowledge Base of Published Experimental Data.* Proceedings of the 5th International Conference on Intelligent Systems for Molecular Biology, 1997: p. 84-87.

38. Altman, R., et al., *RiboWeb: An Ontology-Based System for Collaborative Molecular Biology.* IEEE Intelligent Systems, 1999. **14**(5): p. 68-76.

39. Karp, P. and S. Paley, *Integrated Access to Metabolic and Genomic Data.* Journal of Computational Biology, 1996. **3**(1): p. 191-212.

40. LSID, *http://lsid.sourceforge.net.*

41. Web_service, *http://www.webopedia.com/TERM/Web_services.html.*

42. XML-RPC, *http://www.xmlrpc.com/.*

43. SOAP, *http://www.w3.org/TR/soap/.*

44. myGrid, *http://www.mygrid.org.uk/.*

45. DAS, *http://biodas.org/documents/spec.html.*

46. BPC, *http://lsid.biopathways.org/.*

47. OWL, *http://www.w3.org/TR/owl-ref/.*

48. Swartz, A., *The Semantic Web In Breadth.* 2002.

49. RDF, *http://www.w3.org/RDF/.*

50. Jena, *http://jena.sourceforge.net/.*

12.0 Appendix

I. QueryServlet.java

```
package servlet;

import java.io.*;
import java.sql.*;
import java.util.*;
import java.util.zip.*;

import javax.sql.*;
import javax.naming.*;
import javax.servlet.*;
import javax.servlet.http.*;

import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import org.apache.axis.encoding.XMLType;

import javax.xml.rpc.ParameterMode;

/*****************************
 *
 *Author: Howard Chou
 *Date: 03/17/2005
 *
 *Class: QueryServlet
 *
 *Description: Servlet for translating user query into appropriate SQL
 *             subqueries and queries remote web services associated with the
 *             registered databases. The results returned from the remote
 *             databases are aggregated and stored in the database on the
 *             server.
 *
 *****************************/
public class QueryServlet extends HttpServlet {

    /*****************************
     *
     *Function: init
     *
     *Description: Initializes the servlet.
     *
     *****************************/
    public void init() throws ServletException {
    }

    /*****************************
     *
     *Function: doPost
     *
     *@input  request: servlet request passed through the servlet
     *@input  response: servlet response associated with the servlet
 *
 *Description: Performs the servlet POST operations.
 *
 ****************************/
public void doPost(HttpServletRequest request,
                   HttpServletResponse response)
  throws ServletException,java.io.IOException {
  Context env = null;
  Connection conn = null;

  //Get user query

  String query = request.getParameter("query");
  query = query.trim();

  //Parse query

  String[] queryArray = query.split("\"\\s");

  String[] terms;

  StringBuffer sql = new StringBuffer(" term_id='C25364' OR");

  PreparedStatement pstmt = null;
  PreparedStatement pstmt2 = null;
  PreparedStatement pstmt3 = null;
  PreparedStatement pstmt4 = null;
  PreparedStatement pstmt5 = null;
  PreparedStatement pstmt6 = null;
  PreparedStatement pstmt7 = null;

  ResultSet rs = null;
  ResultSet rs2 = null;
  ResultSet rs3 = null;
  ResultSet rs5 = null;
  ResultSet rs7 = null;

  WeakHashMap filter = new WeakHashMap();
  WeakHashMap tm = new WeakHashMap();
  WeakHashMap tb = new WeakHashMap();
  WeakHashMap tmcd = new WeakHashMap();
  WeakHashMap tbtm = new WeakHashMap();
  WeakHashMap tmtb = new WeakHashMap();
  WeakHashMap dstb = new WeakHashMap();
  WeakHashMap refer = new WeakHashMap();

  HashSet hs;

  int id = 0;

  PrintWriter out = response.getWriter();

  try {
      //Locate JNDI

      env = (Context)new InitialContext().lookup("java:comp/env");
      DataSource pool = (DataSource)env.lookup("jdbc/thes");
```

```
          if (env == null)
            throw new ServletException("No context in QueryServlet");

          if (pool == null)
            throw new ServletException("No data source in QueryServlet");

          conn = pool.getConnection();
          pstmt = conn.prepareStatement("SELECT code FROM terms WHERE
preferred_name=?");

          //Check Cache

          pstmt7 = conn.prepareStatement("SELECT results_id FROM ResultsCache
WHERE query=?");
          pstmt7.setString(1,query);
          rs7 = pstmt7.executeQuery();

          while (rs7.next()) {
            id = rs7.getInt("results_id");
          }

          //Query is not found in cache so perform query writing and
          //querying remote databases

          if (id < 1) {
            //Parse each query term

            for (int i = queryArray.length; i-- > 0;) {
                //Parse filter term to determine which thesaurus terms
                //specified by the user query

                terms = queryArray[i].split("=");
                pstmt.setString(1,(terms[0].replace("\"","")).trim());
                rs = pstmt.executeQuery();

                while(rs.next()) {
                  synchronized(sql) {
                      sql.append(" term_id='");
                      sql.append(rs.getString("code"));
                      sql.append("' OR");
                  }

                  if (terms.length > 1) {
                      if (!filter.containsKey(rs.getString("code"))) {
                        hs = new HashSet();
                      }
                      else {
                        hs = (HashSet)filter.get(rs.getString("code"));
                      }

                      hs.add(terms[1].replace("\"",""));
                      filter.put(rs.getString("code"),hs);
                  }
                }
            }
            //Determine which columns, tables, and databases need to
            //be queried based on which thesaurus terms were queried

            int len = sql.length();

            if (len != 0) {
                String s = sql.substring(0,len-3);
                sql = new StringBuffer();

                synchronized(sql) {
                  sql.append("SELECT term_id,tm_id,tm,tb_id,tb,url,refer FROM
dsinfo WHERE");
                  sql.append(s);
                }

                pstmt2 = conn.prepareStatement(sql.toString());
                rs2 = pstmt2.executeQuery();

                while (rs2.next()) {
                  hs = (HashSet)tmcd.get(rs2.getString("tm_id"));
                  if (hs == null) {
                      hs = new HashSet();
                  }

                  hs.add(rs2.getString("term_id"));
                  tmcd.put(rs2.getString("tm_id"),hs);

                  if (!tm.containsKey(rs2.getString("tm_id"))) {
                      tm.put(rs2.getString("tm_id"),rs2.getString("tm"));
                  }

                  hs = (HashSet)tbtm.get(rs2.getString("tb_id"));
                  if (hs == null) {
                      hs = new HashSet();
                  }

                  hs.add(rs2.getString("tm_id"));
                  tbtm.put(rs2.getString("tb_id"),hs);

                  if (!tb.containsKey(rs2.getString("tb_id"))) {
                      tb.put(rs2.getString("tb_id"),rs2.getString("tb"));
                  }

                  hs = (HashSet)dstb.get(rs2.getString("url"));
                  if (hs == null) {
                      hs = new HashSet();
                  }

                  hs.add(rs2.getString("tb_id"));
                  dstb.put(rs2.getString("url"),hs);

                  hs = (HashSet)refer.get(rs2.getString("refer"));
                  if (hs == null) {
                      hs = new HashSet();
                  }

                  hs.add(rs2.getString("tm_id"));
```

```
                    if (rs2.getString("refer") != null) {
                        refer.put(rs2.getString("refer"),hs);
                    }

                    if (!tmtb.containsKey(rs2.getString("tm"))) {

tmtb.put(rs2.getString("tm_id"),rs2.getString("tb_id"));
                    }
                }
            }

            //Rewrite user query into SQL queries

            Set dstb_keys = dstb.keySet();

            WeakHashMap subqueries = new WeakHashMap();
            WeakHashMap subquery = new WeakHashMap();

            HashSet sel = new HashSet();
            HashSet ds_hs = new HashSet();
            HashSet tb_hs = new HashSet();
            HashSet tm_hs = new HashSet();
            HashSet fl_hs = new HashSet();

            StringBuffer select = new StringBuffer();
            StringBuffer where = new StringBuffer("WHERE");

            String ds_id = null;
            String tb_id = null;
            String tm_id = null;
            String cd_id = null;

            boolean is_ident = false;

            //Get database

            for (Iterator dstb_iter = dstb_keys.iterator();
                 dstb_iter.hasNext();) {
                ds_id = (String)dstb_iter.next();
                ds_hs = (HashSet)dstb.get(ds_id);

                //Get tables associated with database

                for (Iterator ds_hs_iter = ds_hs.iterator();
                   ds_hs_iter.hasNext();) {
                  tb_id = (String)ds_hs_iter.next();
                  tb_hs = (HashSet)tbtm.get(tb_id);

                  //Get columns associated with table

                  for (Iterator tb_hs_iter = tb_hs.iterator();
                       tb_hs_iter.hasNext();) {
                      tm_id = (String)tb_hs_iter.next();
                      tm_hs = (HashSet)tmcd.get(tm_id);

                      //Get thesaurus terms associated with
```

```
//column

for (Iterator tm_hs_iter = tm_hs.iterator();
   tm_hs_iter.hasNext();) {
  cd_id = (String)tm_hs_iter.next();

  //Check if column is associated with
  //an Identifier thesaurus term

  if (cd_id.equals("C25364")) {
      is_ident = true;
  }

  //Check if any of the user query terms
  //contains the thesaurus term

  if (filter.containsKey(cd_id)) {
      fl_hs = (HashSet)filter.get(cd_id);

      //Check filter terms associated
      //with the thesaurus term

      for (Iterator fl_hs_iter = fl_hs.iterator();
         fl_hs_iter.hasNext();) {

        //Check if the thesaurus term
        //is a Sequence thesaurus term

        if (cd_id.equals("C25673")) {
            synchronized(where) {
              where.append(" REGEXP_LIKE(UPPER(\'");
              where.append(fl_hs_iter.next());
              where.append("\'),");
              where.append(tb.get(tb_id));
              where.append(".");
              where.append(tm.get(tm_id));
              where.append(") ");
              where.append("OR");
            }
        }
        else {
            synchronized(where) {
              where.append(" LOWER(");
              where.append(tb.get(tb_id));
              where.append(".");
              where.append(tm.get(tm_id));
              where.append(") LIKE LOWER('%");
              where.append(fl_hs_iter.next());
              where.append("%') OR");
            }
        }
      }
  }
}

//If the query term is associated with
//Identifier, then add it to the new query
```

```
          if (is_ident) {
            synchronized(select) {
                select.append(tb.get(tb_id));
                select.append(".");
                select.append(tm.get(tm_id));
                select.append(",");
            }

            is_ident = false;
            sel.add(tm_id);
          }
      }

    //More query rewrite

      sql = new StringBuffer();

      synchronized(sql) {
          sql.append("SELECT DISTINCT ");
          sql.append(select.substring(0,select.length()-1));
          sql.append(" FROM ");
          sql.append(tb.get(tb_id));
      }

      if (where.length() > 5) {
          synchronized(sql) {
            sql.append(" ");
            sql.append(where.substring(0,where.length()-3));
          }
      }

      //Store query with the identifiers of the
      //selected columns

      subquery.put(sql.toString(),sel);

      //Cleanup

      sel = new HashSet();
      select = new StringBuffer();
      where = new StringBuffer("WHERE");
    }

    //Store queries for each database

    subqueries.put(ds_id,subquery);

    //Cleanup

    subquery = new WeakHashMap();
}

//Rewrite queries

Set subqueries_keys = subqueries.keySet();
Set subquery_keys;
```

```
String query1;
String query2;
String ref;
String ref2;

HashSet sel2;
HashSet queries;
HashSet rqueries = new HashSet();
WeakHashMap subqueries2 = new WeakHashMap();

StringBuffer sb1;
StringBuffer sb2;

int idx1 = 0;
int idx2 = 0;
int idx3 = 0;

//Get query

for (Iterator subqueries_keys_iter = subqueries_keys.iterator();
     subqueries_keys_iter.hasNext();) {
    ds_id = (String)subqueries_keys_iter.next();
    subquery = (WeakHashMap)subqueries.get(ds_id);
    subquery_keys = subquery.keySet();

    queries = new HashSet();

    //Get identifiers of select columns associated
    //with query

    for (Iterator subquery_keys_iter = subquery_keys.iterator();
       subquery_keys_iter.hasNext();) {
      query1 = (String)subquery_keys_iter.next();
      sb1 = new StringBuffer(query1);

      if (!rqueries.contains(query1)) {
          sel = (HashSet)subquery.get(query1);

          //Check if column has a reference

          for (Iterator sel_iter = sel.iterator();
             sel_iter.hasNext();) {
            ref = (String)sel_iter.next();
            hs = (HashSet)refer.get(ref);

            //If a column has a reference to
            //another column, then try to
            //construct new SQL query

            if (hs != null) {
                //Get references

                for (Iterator hs_iter = hs.iterator();
                   hs_iter.hasNext();) {
                  ref2 = (String)hs_iter.next();
```

```
                              for(Iterator subquery2_keys_iter =
subquery_keys.iterator();
                                  subquery2_keys_iter.hasNext();) {
                                  query2 =
(String)subquery2_keys_iter.next();
                                  sel2 = (HashSet)subquery.get(query2);

                                  if (sel2.contains(ref2)) {
                                    sb2 = new StringBuffer(query2);

                                    //Remove referenced
                                    //column from select

                                    idx3 = sb2.indexOf(",");
                                    if (idx3 > -1) {

                                        sql = new StringBuffer();
                                        synchronized(sql) {
                                          sql.append(tb.get(tmtb.get(ref2)));
                                          sql.append(".");
                                          sql.append(tm.get(ref2));
                                        }

                                        idx3 = sb2.indexOf(sql.toString());
                                        if (idx3 > -1) {
                                          len = sql.length();
                                          sb2 =
sb2.replace(idx3,idx3+len,"");

                                          idx3 = sb2.indexOf(" ,");
                                          if (idx3 > -1) {
                                              sb2 = sb2.replace(idx3,idx3+2,"
");

                                          }

                                          idx3 = sb2.indexOf(",,");
                                          if (idx3 > -1) {
                                              sb2 =
sb2.replace(idx3,idx3+2,",");
                                          }

                                          idx3 = sb2.indexOf(", ");
                                          if (idx3 > -1) {
                                              sb2 = sb2.replace(idx3,idx3+2,"
");

                                          }
                                        }
                                    }

                                    //Union queries

                                    idx1 = sb1.indexOf("WHERE");
                                    idx2 = sb2.indexOf("WHERE");

                                    if (idx1 > -1 && idx2 > -1) {
                                        synchronized(sb1) {
                                          sb1.append(" UNION ");
                              sb1.append(sb2);
                            }
                        }

                        if (idx1 > -1 && idx2 < 0) {
                            synchronized(sb2) {
                              sb2.append(" WHERE ");
                              sb2.append(tb.get(tmtb.get(ref2)));
                              sb2.append(".");
                              sb2.append(tm.get(ref2));
                              sb2.append(" IN (");
                              sb2.append(query1);
                              sb2.append(")");
                            }

                            synchronized(sb1) {
                              sb1.append(" UNION ");
                              sb1.append(sb2);
                            }
                        }

                        queries.remove(query2);
                        rqueries.add(query2);
                      }
                  }
                }
            }
          }

          //Add query to new list of database
          //queries only if the query is not a
          //duplicate

          if (sb1.indexOf("WHERE") > -1) {
            queries.add(sb1.toString());
          }
      }
    }

    //Stores new queries associated with each database

    if (queries.size() > 0) {
      subqueries2.put(ds_id,queries);
    }

    //Cleanup

    queries = new HashSet();
}

subqueries = subqueries2;

//Send queries to the remote databases

Service service = new Service();
Call call = (Call)service.createCall();
```

```
            //Decompression variables

            byte[] compressed;
            Inflater decompressor = new Inflater();
            ByteArrayOutputStream bs;
            byte[] buf = new byte[1024];
            int count = 0;

            byte[] decompressed;
            ByteArrayInputStream bs2;
            ObjectInputStream is;

            Object[] ret;

            //Prepare to store results on the server

            pstmt3 = conn.prepareStatement("SELECT Results_seq.NEXTVAL FROM
dual");
            rs3 = pstmt3.executeQuery();
            rs3.next();
            id = rs3.getInt("nextval");

            pstmt4 = conn.prepareStatement("INSERT INTO Results (id,result)
VALUES (?,?)");

            conn.setAutoCommit(false);

            subqueries_keys = subqueries.keySet();

            //Get database

            for (Iterator subqueries_iter = subqueries_keys.iterator();
                 subqueries_iter.hasNext();) {
                ds_id = (String)subqueries_iter.next();
                ds_hs = (HashSet)subqueries.get(ds_id);

                out.println(ds_id);
                out.println(ds_hs.toString());

                //Get queries for each database

                for (Iterator ds_hs_iter = ds_hs.iterator();
                   ds_hs_iter.hasNext();) {

                  //Initialize remote web service

                  call.setTargetEndpointAddress(new java.net.URL(ds_id));
                  call.setOperationName("query");

                  //Decompress result returned from web service

                  compressed = (byte[])call.invoke(new Object[]
{(String)ds_hs_iter.next()});
                  decompressor.setInput(compressed);
                  bs = new ByteArrayOutputStream(compressed.length);

                  buf = new byte[1024];
                while (!decompressor.finished()) {
                    count = decompressor.inflate(buf);
                    bs.write(buf,0,count);
                }

                bs.close();
                decompressor.reset();

                decompressed = bs.toByteArray();
                bs2 = new ByteArrayInputStream(decompressed);
                is = new ObjectInputStream(new BufferedInputStream(bs2));
                ret = (Object[])is.readObject();

                bs2.close();
                is.close();

                //Store results in database

                out.println(ret.length);

                for (int i = ret.length; i-- > 0;) {
                    tm_id = (String)ret[i];

                    if (tm_id != null && tm_id.length() > 0) {
                      pstmt4.setInt(1,id);
                      pstmt4.setString(2,tm_id);
                      pstmt4.addBatch();
                    }
                }
              }
          }

          pstmt4.executeBatch();

          //Cache query

          pstmt6 = conn.prepareStatement("INSERT INTO ResultsCache
(results_id,query) VALUES (?,?)");
          pstmt6.setInt(1,id);
          pstmt6.setString(2,query);
          pstmt6.executeQuery();

          conn.commit();

          conn.setAutoCommit(true);


          //Cleanup

          decompressor.end();
        }

        //Get number of results

        int total = 0;
```

```
          pstmt5 = conn.prepareStatement("SELECT COUNT(*) AS cnt FROM results
WHERE id=?");
          pstmt5.setInt(1,id);
            rs5 = pstmt5.executeQuery();
          rs5.next();
          total = rs5.getInt("cnt");

          //Store information about results in session

          HttpSession session = request.getSession();

          session.setAttribute("resultsId",id);
          session.setAttribute("resultsTotal",total);
      } catch (Exception e) {
          throw new ServletException(e.getMessage());
      } finally {
          try {
            if (rs != null)
                rs.close();

            if (rs2 != null)
                rs2.close();

            if (rs3 != null)
                rs3.close();

            if (rs5 != null)
                rs5.close();

            if (rs7 != null)
                rs7.close();

            if (pstmt != null)
                pstmt.close();

            if (pstmt2 != null)
                pstmt2.close();

            if (pstmt3 != null)
                pstmt3.close();

            if (pstmt4 != null)
                pstmt4.close();

            if (pstmt5 != null)
                pstmt5.close();

            if (pstmt6 != null)
                pstmt6.close();

            if (pstmt7 != null)
                pstmt7.close();

            if (conn != null)
                conn.close();

            if (env != null)
```

```
                env.close();
            } catch (Exception e) {
              throw new ServletException(e.getMessage());
            }
        }

        //Redirect page to display query results

//      response.sendRedirect("query-data-sources-2.jsp?page=1");
      }

}
```

II. DbQueryService.jws

```
import java.io.*;
import java.sql.*;
import java.util.zip.*;
import java.util.HashSet;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.sql.DataSource;

/*****************************
 *
 *Author: Howard Chou
 *Date: 03/17/2005
 *
 *Class: DbQueryService
 *
 *Description: Web service interface for querying database associated
 *with this service.
 *
 *****************************/
public class DbQueryService {

    /*****************************
     *
     *Function: query
     *
     *@input  _queryString:SQL query string
     *@output byte array representation of the query result after
     *        compression
     *
     *Description: Passes the query string to the database and returns
     *             the result returned by the databse.
     *
     *****************************/
    public byte[] query(String _queryString) {
      Context env = null;
      Connection conn = null;

      PreparedStatement pstmt = null;
      ResultSet rs = null;

      byte[] compressed = null;

      try {
          //Locate JNDI

          env = (Context)new InitialContext().lookup("java:comp/env");
          DataSource pool = (DataSource)env.lookup("jdbc/swissp");

          if(env == null)
```

```
            throw new Exception("No context available in DbQueryService");

          if(pool == null)
            throw new Exception("no data source available inn
DbQueryService");

          //Get database connection from connection pool and query
          //database

          conn = pool.getConnection();
          pstmt = conn.prepareStatement(_queryString,
                              ResultSet.TYPE_SCROLL_INSENSITIVE,
                              ResultSet.CONCUR_READ_ONLY);
          pstmt.setMaxRows(200);

          rs = pstmt.executeQuery();

          //Get number of rows

          rs.last();
          int numRows = rs.getRow();
          rs.beforeFirst();

          //Get number of columns

          ResultSetMetaData rsm = rs.getMetaData();
          int numCols = rsm.getColumnCount();

          //Aggregate results from query into a hash set

          HashSet hs = new HashSet(numRows);
          numCols++;
          for(int i = numRows; i-- > 0;) {
            rs.next();

            for (int j = numCols; --j > 0;) {
                hs.add(rs.getObject(j));
            }
          }

          //Compress hash set with results

            ByteArrayOutputStream bs = new ByteArrayOutputStream(5000);
            ObjectOutputStream os =
            new ObjectOutputStream(new BufferedOutputStream(bs));

            os.flush();
            os.writeObject(hs.toArray());
            os.flush();

            byte[] send = bs.toByteArray();

            Deflater compressor = new Deflater();
            compressor.setLevel(Deflater.BEST_COMPRESSION);
            compressor.setInput(send);
            compressor.finish();
            ByteArrayOutputStream bs2 = new
ByteArrayOutputStream(send.length);

          int count = 0;
            byte[] buf = new byte[1024];
            while (!compressor.finished()) {
                count = compressor.deflate(buf);
                bs2.write(buf,0,count);
            }

          //Store result

            compressed = bs2.toByteArray();

          //Cleanup

          bs.close();
            os.close();

            bs2.close();
          compressor.end();
      } catch(Exception e) {
      } finally {
          try {
            if(rs != null)
                rs.close();

            if(pstmt != null)
                pstmt.close();

            if(conn != null)
                conn.close();

            if(env != null)
                env.close();
          } catch(Exception e) { }
        }

      //Return compressed result

      return compressed;
    }

}
```

6650 · 110