DESIGNING MIDDLEWARE TO FACILITATE ANALYSIS OF DISPARATE ENVIRONMENTAL DATASETS

Wesley Leonard

A thesis submitted in partial fulfillment of
the requirements for the degree of
Master of Science

Department of Computer Science

Central Michigan University
Mount Pleasant, Michigan
October, 2008

Accepted by the Faculty of the College of Graduate Studies,

Central Michigan University, in partial fulfillment of

the requirements for the master's degree

Thesis Committee:

_____ Committee Chair

_____ Faculty Member

_____ Faculty Member

Date: _____

_____ Dean
College of Graduate Studies

Date: _____

Committee:

Paul Albee, Ph.D., Chair

Michael Stinson, Ph.D.

Tracy Galarowicz, Ph.D.

This is dedicated to my supportive
and patient wife, Cynthia Drake.

ACKNOWLEDGMENTS

This thesis would not be possible without the support of the Computer Science Department at Central Michigan University and the guidance of my committee: Dr. Paul Albee, Dr. Michael Stinson, and Dr. Tracy Galarowicz. I would also like to thank Thomas Rohrer, director of the Environmental Studies Program at CMU, and the Michigan Department of Environmental Quality. Finally, I wish to acknowledge the support of Central Michigan University in producing this work.

ABSTRACT

DESIGNING MIDDLEWARE TO FACILITATE ANALYSIS
OF DISPARATE ENVIRONMENTAL DATASETS

by Wesley Leonard

A middleware system to facilitate the exploration, combination, organization, conversion, and analysis of data sets was designed and built. The web-based system provides management controls, supports most operating environments, supports many data sources, provides incremental or full data extracts, provides transformation of source data, provides data cleansing, and supports custom SQL for browsing or exporting data. Methods of caching data from remote sources and joining tables from different databases were designed and evaluated. After processing, data from remote sources or local cache tables are exported in XML format. The prototype system was built with Ruby on Rails. A database of fish contaminant studies from the Michigan Department of Environmental Quality was used extensively as a test dataset.

The algorithms presented in this thesis are used to create a computationally efficient middleware system to support data analysis. The data caching technique developed in this thesis supports joins across heterogeneous database platforms, and is shown to improve data retrieval and transformation performance.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF LISTINGS

CHAPTER I

INTRODUCTION

This thesis addresses the problem of storing and managing large sets of environmental data from different sources. Specifically, fish contaminant data collected by the Michigan Department of Environmental Quality needed to be stored, organized, and accessed by researchers and the public. Also, datasets from other organizations, in different locations, or in different formats needed to be combined with the fish contaminant dataset for analysis. Software tools needed to be created to facilitate the organization, combination, and export of these datasets for analysis. Furthermore, the data needed to be available over the web via a simple interface for browsing and configuring datasets and through a web service for export. Through the web service it should be possible to join tables from different datasets, perform data transformations, and make the data readily accessible to analysis software.

Background

Since 1980, the Michigan Department of Environmental Quality has been collecting data on the concentration of certain contaminants (including mercury and PCBs) in fish throughout the state of Michigan [4]. This dataset contains 29,033 samples from 28,586 different fish. Sixty-eight different species of fish are represented and three hundred twenty-six different types of contaminants are measured. As an independent study project in 2006, a system was developed to store this information in a PostgreSQL database and provide a flexible web front-end to view and analyze the data. All software used to build this system was open source and freely available.

This system utilizes the R Environment for data analysis. R is an integrated suite of software facilities for data manipulation, calculation, and graphical display. R offers data

handling and storage, operators for calculations, a large collection of intermediate tools for data analysis, graphical analysis tools, and a simple programming language [22]. For this application, users may run pre-existing R programs or create custom R code to analyze the available data. See Figures 1 and 2 for examples of this preliminary work with this data.

Figure 1 shows the web interface to the "fishR" application. The user is presented with two predefined analyses: *Average Mercury Concentration for all Species by Year* and *Average PCB Concentration for all Species by Year*. Figure 2 shows the plot that results from running the first analysis (mercury concentration). The user may alter the R code presented in the text boxes to change which tables in the source database are read, which fields are read, how data analyzed, and what information (including textual, numeric, or graphical) is displayed.

Using this tool, a user would need to be familiar with R to explore and analyze the dataset effectively. It was believed that a tool that could bring this and other datasets into common, user-friendly analysis tools would allow researchers to take advantage of this valuable dataset. This idea led to the work performed for this thesis.

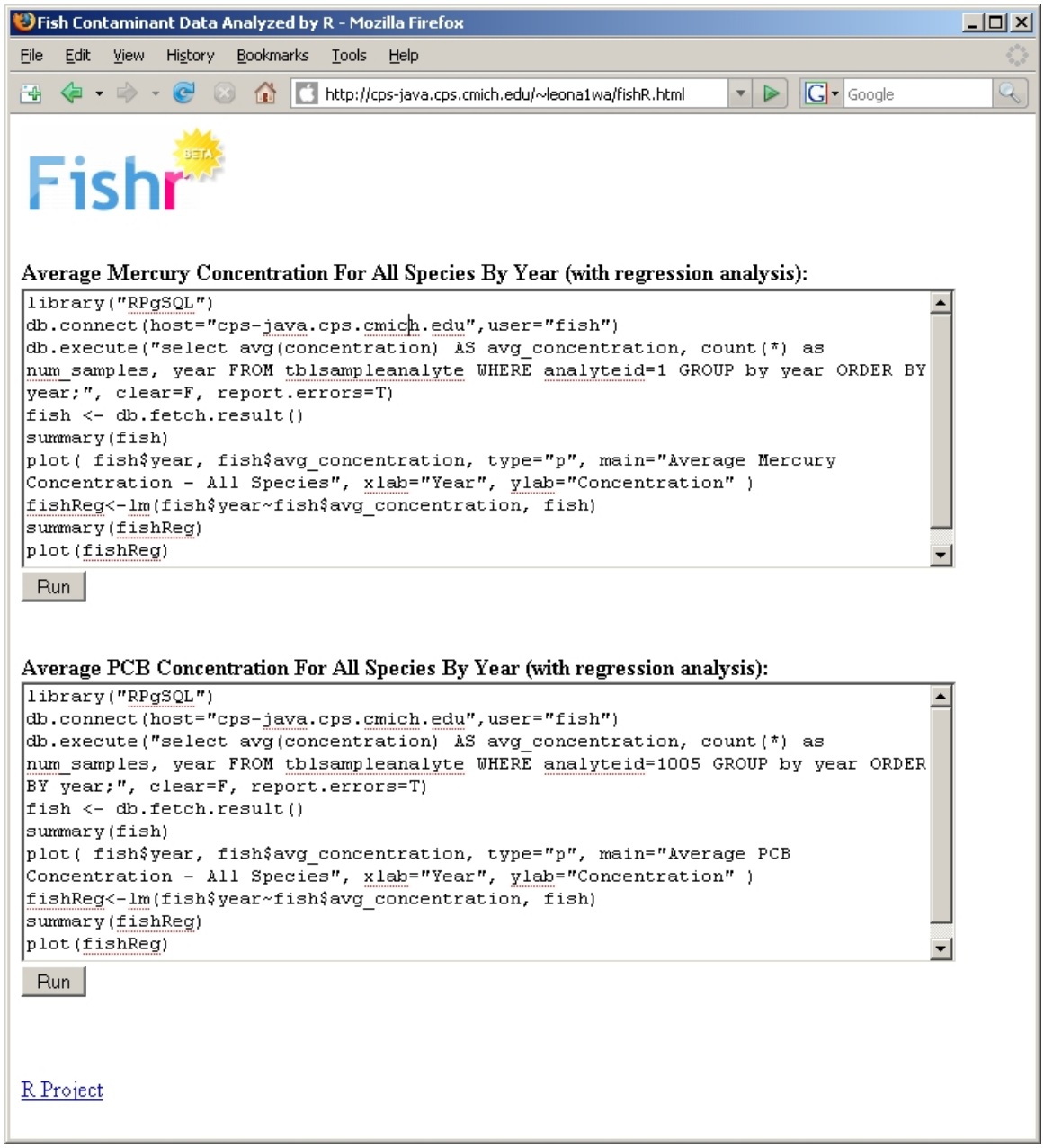

Figure 1. Example of preliminary work - web front end for R

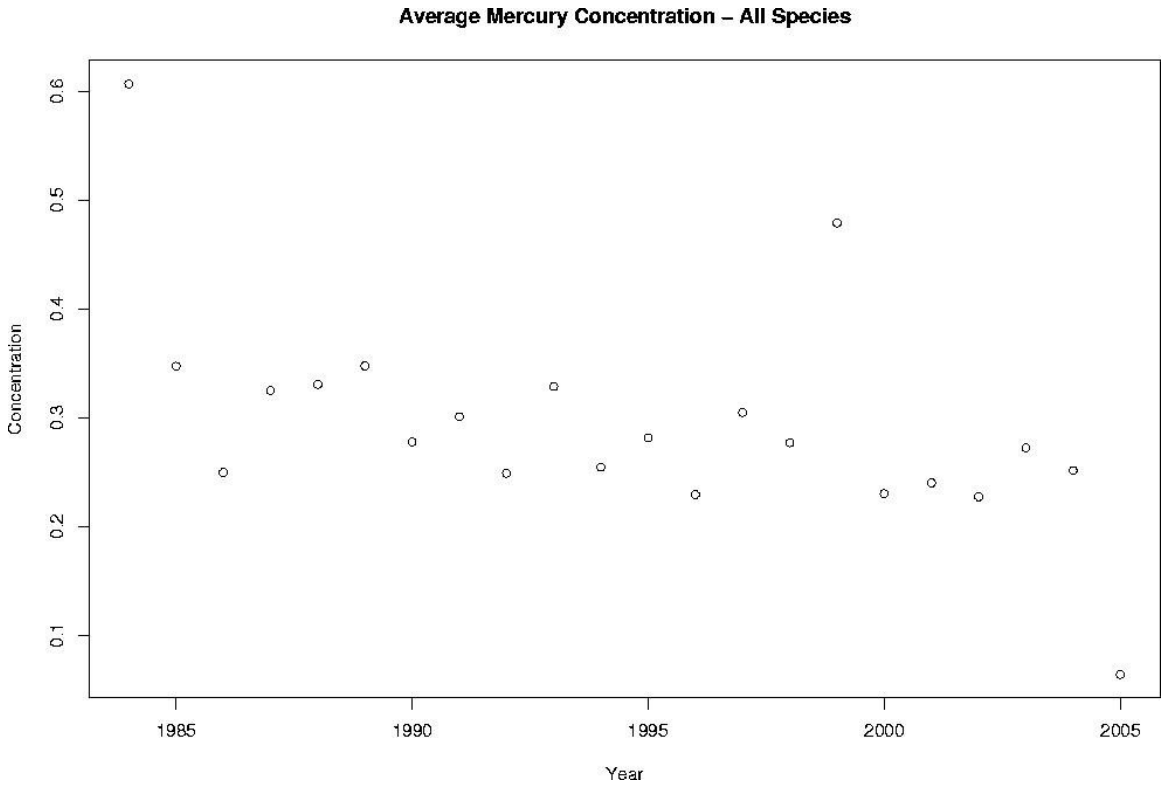

Figure 2. Plot of mercury concentration for all species

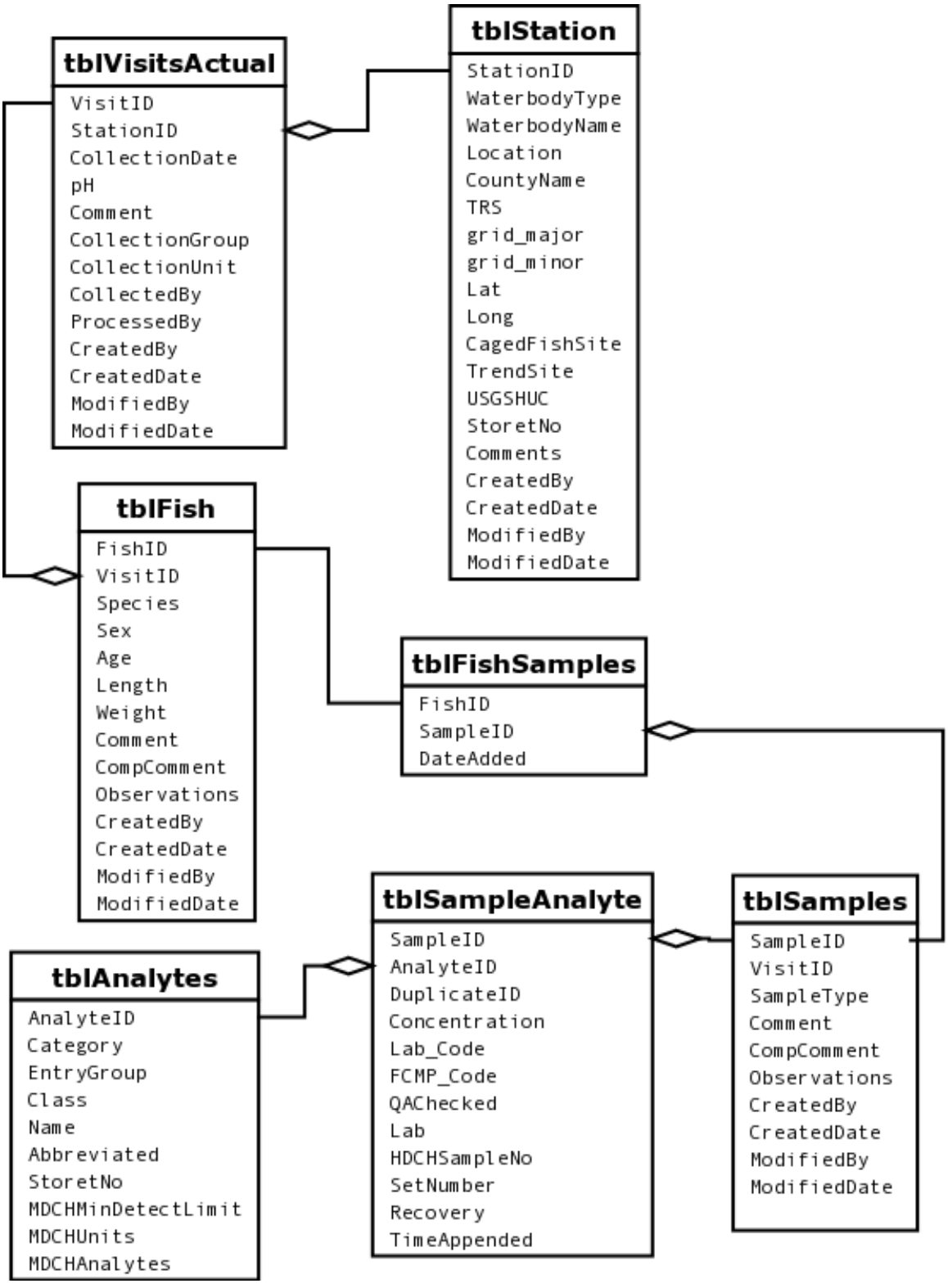

Figure 3. Structure of MI DEQ fish contaminant database

CHAPTER II

LITERATURE REVIEW

The vast and ever increasing amount of scientific and business data available has made the use of computer-based data mining tools a necessity. As the use of modern data collection tools (such as automated sensor networks) increases, so will the need for more sophisticated and complex analysis tools [14]. This "data glut" has been growing since the 1990s, making traditional methods of data analysis infeasible [18].

Description of Selected Works

Motivation and Similar Work

An editorial published in "Issues in Science and Technology" by H. Spencer Banzhaf describes how better data collection and analysis can help improve environmental policy and enforcement. As an example of why resources should be put toward the analysis of environmental and scientific data, the author describes how the analysis of historical economic data has made it possible to set economic policies in a way that avoids negative trends such as recession and inflation [3].

A system similar to the one proposed in this thesis was described in a paper titled "Development of a Data Mining Application for Huge Scale Earth Environmental Data Archives" by Ikoma et al. This system was designed to integrate and analyze 900 GB of data from various sources. Some of the data sources included long-wave radiation reading provided by the National Oceanic and Atmospheric Administration, wind speed data from the National Center for Environmental Prediction/National Center for Atmospheric Research, and seasonal cloud change data provided by the International Satellite Cloud Climatology Project. The designers of this system sought to provide a more user-friendly environment than existing analysis tools offered [10].

Data Mining and Analysis

Data mining is the process of sorting through large amounts of data and picking out new, valuable, and nontrivial information. The two primary goals of data mining are prediction (using variables within the dataset to predict unknown or future values of other variables) and description (finding patterns within data that can be interpreted by humans) [14].

While an understanding of data mining and analysis techniques is important, this thesis focuses on the preliminary steps in data mining and data analysis. Selection, preparation, and preprocessing of raw datasets is a necessary step in the process of data mining. Outlier detection is one common preprocessing task which reduces the amount of data available for analysis by identifying and possibly removing unusual values that could result from measurement or recording errors and may drastically affect the analysis of the data. Other common tasks include scaling, encoding, and selecting features (making sure that data values with different ranges are appropriately weighted for analysis) [14, 15].

Often scientists wish to analyze data from different sources in order to cross-validate their findings; however, scientific data often needs preprocessing to convert it from its initial format into a format more accessible to analysis tools. Raw scientific data may have missing values that need to be filtered out or extrapolated or high-dimensional data that must be simplified for analysis. Scientific data frequently has both spatial and temporal aspects (such as sensor data, which is collected in one location over a period of time). Also, data that has been generated manually may be incomplete or labeled improperly [23].

Data Warehousing

A data warehouse can be thought of as a read-only database that creates a single logical view of data and is accessed through a front-end tool or application [18]. The primary purpose of this database is analysis and it is kept separate from operational databases.

This separation is necessary because the performance of an operational database may suffer if it were to share system resources with a data warehouse. Also, it is often impossible to structure a database in such a way that it could perform adequately for both analytical and operational purposes [13]. A traditional operational database is application-oriented, usually not integrated with other databases, continuously updated, may contain only current (not historical) data, and is accessed in a predictable manner. A data warehouse, in contrast, is subject-oriented, may be made up of several integrated databases, nonvolatile, contains stabilized data values, and is accessed in an ad hoc fashion [5].

The field of data warehousing is not mature, so there are many different methodologies being used. Some methodologies are linked to a particular vendor's database system (such as Oracle, IBM DB2, Sybase or Microsoft SQL Server), while others are based on a particular infrastructure or information modeling tool (such as SAS, SAP, or PeopleSoft) [17].

A two volume eBook titled "Encyclopedia of Data Warehousing and Mining" contains several short articles on concepts including classification methods, data warehouse back-end tools, and storage strategies in data warehousing. One article titled "A General Model for Data Warehouses" proposes a model to define a well-formed data warehouse structure. The method described by Michel Schneider describes how facts and dimensions can be modeled in an acyclic graph structure (DWG). The DWG has several roots and different paths from the roots can always be divided into two sub-paths (one with only fact nodes and another with only dimension nodes) [20, 21].

A data mart is similar to a data warehouse, but it contains a narrow scope of data (a single subject, business function, or application). Data marts are often implemented on second-tier servers, which draw data from a centralized warehouse and deliver the cleansed data to clients [5]. Dimensional modeling is a typical data mart modeling technique. Using this technique, facts (usually numerical) are stored in one or more tables and dimensions

(textual descriptions related to facts) are stored in one or more additional tables. The intent is to represent information in a way that allows answers to be quickly retrieved from the dataset [13]. For example, a facts table that has only numeric identifiers and contaminant concentration values could be used to quickly retrieve the numeric identifiers that have concentrations of a contaminant over a certain threshold. A dimension table could contain information about the sample (based on the numeric identifier), such as the species of fish, date it was analyzed, and location where the sample was taken.

Metadata is another important part of a data warehouse. Metadata is defined as data about data or information required to make data useful. Traditional database design uses a schema to describe the conceptual or logical data structure of entities in a database as well as their relationships. The distinction between data and metadata is simply how the data is used. Metadata makes it possible to understand what data is available, how to access it, and how to interpret it. Metadata may contain the following information: [12, 18]

- A data dictionary (definitions of the databases being maintained and the relationships between elements)
- Data flow (direction and frequency of data feed)
- Data transformation
- Version control
- Data usage statistics
- Column or attribute alias information
- Security

Middleware

The purpose of middleware is to establish a software layer that homogenizes an infrastructure by means of a well-defined and structured programming model [11]. In a data warehouse, the architectural objectives of middleware include support for:

- A wide range of data sources
- The propagation of both incremental changes and full extracts
- Source data transformations
- Data cleansing
- Documented SQL and other open interfaces for third-party and user-defined code
- Targets and sources with unpredictable connectivity [5]

Middleware provides general access to data and makes it possible for heterogeneous databases to be connected together in a single view. Middleware technology designed to enable the implementation of a data warehouse must do the following:

- Incorporate controls and management tools to manage an enterprise-wide view of data
- Provide a single database, operating system, and network-independent interface to the user and developer
- Provide an easy-to-use API (Application Programming Interface) which can be embedded into applications
- Support a wide variety of data managers and operating environments

In a data warehouse, middleware's key role is isolating applications from infrastructure. Physical databases are part of this infrastructure. With proper middleware, a data warehouse acts as a single, consistent interface to the database infrastructure [18].

A paper by El Maghraoui et al. describes the Internet Operating System (IOS): an agent-based middleware system for effectively utilizing internet-wide grid resources [9]. This paper describes architectural features of middleware, programming models, and application issues that are relevant to this thesis. The use of grid-based computing is an emerging trend with the potential to change the way information is collected, shared, and analyzed.

Web Services and XML

Web services refer to a set of technologies that enable networked and modular applications. Such technologies include Simple Object Access Protocol (SOAP), Web Services Definition Language (WSDL), and Universal Description, Discover, and Integration (UDDI) protocol [24, 8]. The web services framework has been developed to offer applications as services both within an organization and externally. This service-oriented computing paradigm is built upon open XML-based standards [7].

XML is a self-describing semistructured data format. XML documents used for data exchange can be sent as static documents or generated dynamically by computer programs. Milo et al. refer to XML documents where parts of the data are generated by computer programs as "intensional documents". The term "materialization" refers to the process of evaluating a program in an intensional XML document and replacing it with the result of the program. This process is similar to the way in which Ruby, PHP, or other scripting languages can be embedded in HTML and evaluated by a web server when the document is requested [16].

Use of XML and web services provide a framework for interoperability between solutions. Web services help achieve flexible, secure, and coordinated resource sharing among systems. This makes it possible to take advantage of distributed data collection, data analysis, and other emerging "grid computing" technologies. This new paradigm is referred to as the "semantic web" [8].

Ruby on Rails

Ruby on Rails was introduced in 2004 and has gained a reputation for being a quick framework for developing web applications [19]. Rails applications use the Model-View-Controller (MVC) architecture, in which the application's data, business logic, and user interface are isolated from each other. Rails comes bundled with several built-in classes to abstract such tasks as database access and web interface construction. Ruby, the underlying language, is an entirely object-oriented programming language based on Perl [6].

Conclusions

The literature shows a clear need for automated tools to store, organize, and facilitate the analysis of data. Research continues to improve the tools and techniques used for storing and organizing information (data warehousing), sharing information between different system (the semantic web), analyzing data, and building interfaces to datasets.

CHAPTER III

THESIS STATEMENT

A middleware system to facilitate the exploration, combination, organization, conversion, and analysis of data sets will be designed and a prototype will be built. The design and implementation of this system will be based on current research in the areas of data mining, data warehousing, middleware, and information sharing through web services. Research into these topic revealed the following criteria, which will be used to gauge the effectiveness of the tool and its ability to contribute significantly to contemporary research [5, 11]:

- Provide controls and management tools to access and view data
- Support a wide variety of operating environments
- Support a wide range of data sources
- Support the propagation of both incremental changes and full extracts
- Support source data transformations
- Support data cleansing (removing inconsistent records from the data set)
- Support documented SQL and other open interfaces for third-party and user-defined code

Description of System Components

The system designed for this thesis will consist of the following primary components: a relational database for storage of cached data and configuration information, a web-based data import and management interface, a middleware layer to handle database

connections and other tasks related to low-level data processing, and a web service interface to export processed data to analysis tools in XML format.

The relational database will store metadata that has been generated from external data sources, processed data that has been cached, data transformation settings, data export settings, and user login information. PostgreSQL, an open-source object-relational database system based on the University of California, Berkeley's POSTGRES [1], was selected as the database server for this thesis. PostgreSQL is available at no cost, operates on many different server platforms, and has a feature set comparable to commercial enterprise database servers. Also, PostgreSQL is being used successfully by several companies and research organizations (such as the U.S. Centers for Disease Control).

The web-based management interface will provide a user-friendly interface to the middleware functions, configuration database, imported datasets, data transformations, and exported datasets. This interface will provide tools for managing access to the system (creating users and changing user passwords), importing data from various sources, previewing datasets, exporting data, and managing caching and transformation of source data while maintaining compatability with all modern web browsers and operating systems.

The middleware component will provide underlying functionality to the management interface and web service interface by handling database connections, authenticating user logins, managing configuration information, caching data, and applying data transformations.

The web service interface will make data available in XML format through a unique URI. Exported data may be taken from one or more datasets and transformed or combined as configured by the user.

Ruby on Rails was chosen as the development environment for the web interface, middleware component, and web services interface. Ruby on Rails offers a powerful, object-oriented programming language and a modern approach to rapid application devel-

opment. Also, Rails provides built-in support for many modern database systems, including MySQL, Microsoft SQL Server, and PostgreSQL.

Definition of Terms

The following terms as used in this thesis may differ from their common definition or have multiple accepted meanings depending on context. To avoid ambiguity, the terms below will be used throughout this thesis as described in this section.

- Dataset – A collection of data. This may be composed of one or more relational database tables. For this thesis, it is assumed that the dataset is stored in some form of relational database.

- Export – A dataset made available to an external application for analysis as configured by the user. The system will not limit the number of fields or records exported. An export may be made up of several source datasets that have been processed by the middleware component.

- Exported field – A particular field from a table in a dataset that is selected for export. An exported field may be exported under its original name or a new name. (If the user wishes to combine similar data from two sets it would be advisable to use uniform field names for similar data). Transformations may be applied to an exported field.

- Transformations (or Transforms) – Changes applied to exported fields. Transformations may be necessary to clean up data, properly filter out inappropriate values, truncate text fields, or scale numeric fields.

- SQL – Structured Query Language. Common, but not necessarily standard, language used by many database engines for viewing, updating, inserting, and deleting data. SQL can also be used to create or alter the schema of a relational database. When

SQL is referenced throughout this thesis, it is strictly referring to the subset of SQL common to most modern database engines (such as PostgreSQL, Microsoft SQL Server, Oracle, and MySQL).

CHAPTER IV

DESCRIPTION OF ALGORITHMS

The purpose of this thesis is to design a system to explore, combine, organize, transform, and export datasets. The ability to access data from disparate datasets through a unified interface is essential to this goal. To accomplish this, a simple but effective method of reading data from each data source was designed and implemented. This method was expanded to support table joins and save computation and transmission time by caching processed data.

Reading Data

Initially, the system was designed to read data from source databases using the following simple method:

Algorithm 1: Simple data access method

(1) **foreach** $d \in datasets$

(2) **foreach** $t \in d_{tables}$

(3) $query \leftarrow$ GENERATE_QUERIES$(export)$

(4) $results \leftarrow results \cup$ EXECUTE$(query)$

(5) RENDER_XML$(results)$

Queries are generated based on which fields and tables are configured for export from remote datasets. In lines 1-3 in algorithm 1, each dataset configured for export is examined and a query is generated to read the exported fields and tables from the remote data source. On line 4 the query is executed on the remote database and the resulting records are stored in the result set. After all queries have been executed, the entire result set is rendered in XML for display (line 5). (Details of connecting to remote databases have been left out of this section as they are not essential to understanding the algorithms used

for data extraction. A thorough explanation of how database connectivity between multiple remote sources was accomplished can be found in the **Implementation Details** chapter).

This method performed adequately for simple exports with no transformations, but it had limitations. Using this method, it was not possible to join tables from different data sources. Implementing joins between tables on the same database could have been implemented by making a simple modification to the query generation function. Joining tables from different databases could not be achieved as easily. Another drawback to this method is that transformations must be applied each time the export is read, which may be quite CPU-intensive and slow. It became clear that some form of data caching was necessary to perform joins on tables from different databases and improve performance when data transformations are used.

Data Caching

Caching of remote data is clearly necessary in order to accomplish the goals set forth in this thesis. Joining two tables from datasets on different servers would not be possible without caching. (Joining databases on different servers from the same vendor may be possible without caching by using proprietary functionality. While this would work for a limited set of database severs it is not an inclusive or acceptable solution for this thesis). Also, since responsiveness and availability of remote database servers can vary, caching data improves response time and reliability while reducing the load on the network and remote database servers.

The caching scheme used for this thesis resembles simple replication. Only the fields requested from the external data source are stored in a cache table locally and any query requiring those fields can simply query the cached data, provided it has not expired or become corrupt. Also, querying cached data requires no conditions or transformations as these are handled prior to caching. This technique is known as sub-table caching [2].

Algorithm 2: Dynamic merged cache/query algorithm

(1) **if** $Cache_{config} \wedge (Cache_{expired} \vee Cache_{dirty})$
(2) $\quad Cache_{update} \leftarrow true$
(3) **if** $\neg Cache_{config} \wedge J \neq \emptyset$
(4) $\quad Cache_{update} \leftarrow true$
(5) **if** $Cache_{config} \vee J \neq \emptyset$
(6) $\quad Cache_{read} \leftarrow true$
(7) $queries \leftarrow$ GENERATE_QUERIES$(export)$
(8) $T_c \leftarrow \emptyset$
(9) **foreach** $q \in queries$
(10) $\quad$ **if** $\neg Cache_{read} \vee Cache_{update}$
(11) $\quad\quad results \leftarrow$ EXECUTE(q)
(12) $\quad$ **if** $Cache_{update}$
(13) $\quad\quad$ STORE_IN_CACHE_TABLE(results)
(14) $\quad\quad T_c \leftarrow T_c \cup$ CACHE_TABLE_NAME(q)
(15) $\quad$ **else if** $\neg Cache_{read}$
(16) $\quad\quad result\_set \leftarrow results$
(17) $\quad$ **else if** $Cache_{read}$
(18) $\quad\quad T_c \leftarrow T_c \cup$ CACHE_TABLE_NAME(q)
(19) **if** $Cache_{read}$
(20) $\quad$ **if** $J \neq \emptyset$
(21) $\quad\quad$ **foreach** $j \in J$
(22) $\quad\quad\quad result\_set \leftarrow result\_set \cup j_{table1} \bowtie j_{table2}$
(23) $\quad\quad\quad T_c - \{j_{table1}, j_{table2}\}$
(24) $\quad$ **foreach** $t \in T_c$
(25) $\quad\quad result\_set \leftarrow result\_set \cup$ QUERY_CACHE_TABLE(t)
(26) RENDER_XML$(result_set)$

The algorithm used for this thesis merges caching and querying of data. This will be referred to as the dynamic merged cache algorithm, or simply DM. The DM algorithm first determines if caching is used at all (based upon the configuration of the exported dataset and whether joins are included in the export) and if the cached data needs to be refreshed (based upon the expiration date and estimated consistency of cached data).

Algorithm 2 illustrates the DM algorithm. Lines 1 through 6 examine the configuration and state of the cached data to determine how to set the two control variables which determine the execution plan. These variables are $Cache_{update}$ and $Cache_{read}$.

Algorithm 3: Algorithm to generate queries of source datasets
GENERATE_QUERIES($export$)

(1) $Q \leftarrow \emptyset$

(2) $D \leftarrow$ GET_DATASETS$(export)$

(3) **foreach** $d \in D$

(4) $T \leftarrow$ GET_TABLES_EXPORTED(d)

(5) **foreach** $t \in T$

(6) $q \leftarrow$ "SELECT t_{fields} FROM t"

(7) **if** $Cache_{update} \wedge Cache_{type} = incremental$

(8) **if** $isset(table_{key\_field})$

(9) $max \leftarrow$ MAX$(Cache_{key_field})$

(10) $q \leftarrow q +$ " WHERE key_field $> max$"

(11) $Q \leftarrow Q \cup q$

(12) **return** Q

$Cache_{update}$ indicates whether the cache requires updating. This is initialized as false (not shown in the algorithm) but can be set to true on line 2 if the exported dataset is configured to use caching ($Cache_{config}$ is true) and one of the following conditions are also true: the cache has expired ($Cache_{expired}$ is true) or the cache has become inconsistent ($Cache_{dirty}$ is true). $Cache_{expired}$ is true if the expiration date of the cache has passed. $Cache_{dirty}$ is true if the cache is potentially inconsistent. A "dirty" or inconsistent cache occurs when the user makes configuration changes to the export. By adding fields, removing fields, adding joins, or adding transforms, the user changes the expected output of the export. The information already in the cache tables must be replaced. $Cache_{update}$ will also

be set to true if joins are used and caching is not configured (lines 3-4). Joins require the use of cache and it is assumed that no pre-existing cached data is available if caching has not been configured for this export.

$Cache_{read}$ indicates whether the export is read from cache (as opposed to read directly from the source database). This is initialized as false and will be set to true if the export is configured to use caching ($Cache_{config}$ is true) or the export contains any joins (lines 5-6). As explained above, joining tables requires the use of caching even if caching is not configured for the export. (In the situation where joins are used and caching is not configured, the cache is generated to perform the export and is immediately discarded. Subsequent reads of the exported dataset will still read data from the source databases, cache the data, perform joins, and proceed to discard the cached data). In the DM algorithm, joins are stored in the set J. When J is not empty ($J \neq \emptyset$) then one or more joins have been configured for the export. Populating the set J is accomplished by simply reading join details from a configuration table (not covered in the algorithm).

Line 7 calls a function to generate queries based upon the export configuration. (The simple algorithm presented earlier uses this same method). This function returns a set of all queries necessary to read data from remote data sources. Algorithm 3 illustrates the steps involved in query generation. This algorithm will be discussed in detail later in this chapter.

Line 8 simply initializes the set of cache tables (T_c) to the set. The set of cache tables contains the names of local cache tables which may contain data to be exported. The cache table names are unique and based upon the names of the corresponding data tables from remote sources.

Line 9 begins the process of reading external (or cached) data by iterating through the set of generated queries. If caching is not used ($Cache_{read}$ is false) or the current cache needs to be updated ($Cache_{update}$ is true), the query is executed and the results are stored

in a temporary container called *results* (lines 10-11). Lines 12-18 present three mutually exclusive possibilities. First, if a cache update is required, the results of the previously executed query are stored in a cache table and that table is added to the set of available cache tables (lines 12-14). If a cache update is not required (the condition on line 12 is false) and the export will not be read from cache, then the results of the query are simply added to the final result set (lines 15-16). Finally, if this export is to be read from cache, the cache table associated with this query is stored in the set of available cache tables (lines 17-18). It should be noted that all operations performed in lines 10-18 occur for each of the queries generated for the export.

For brevity, the application of data transformations was not shown in the DM algorithm. Transforms are applied after remote database queries are executed and before the results are stored in cache tables (or added to the final result set). Resulting records are transformed simply by using the value of the field being transformed as the argument to the defined transformation function. The original value is then replaced by the result of the function.

After iterating through all queries, reading from remote data sources, and updating cache if necessary, line 19 determines if the results are to be read from cache. (If caching is not needed, then lines 19-25 are skipped altogether). If the export will be read from cache, line 20 determines if there are any joins configured for this export. If joins are used then the algorithm iterates through each configured join (line 21), performs the join on cache tables and stores the results in the final result set (line 22), and removes the joined tables from the set of available cache tables (line 23).

All remaining cache tables are read and added to the result set in lines 24-25. Joined tables are removed from the set of available cache tables before this step to avoid data duplication.

The final step in the DM algorithm is to render the complete result set in XML.

Analysis of the DM Algorithm

The speed of the DM algorithm (worst case) can be determined by identifying the execution steps and conditions as follows:

$$T_{gq} + N_q * Q * N_r * (T + S_c) + N_j * Q_{cj} + (N_q - 2 * N_j) * Q_c * N_{cr} * S$$

Where T_{gq} is the time to generate queries, N_q is the number of queries, Q is the time to execute a single query, N_r is the number of results, T is the time to apply data transformations, S_c is the time to store the result record in a cache table, N_j is the number of joins, Q_{cj} is the time to execute a join query on cached tables, Q_c is the time to execute a query on a cache table (without joins), N_{cr} is the number of results from the cache table, and S is the time to store results in the result set.

The time to generate queries is negligible and the number of queries will likely be small. Time to execute queries on remote datasets can vary greatly, but due to the small number of queries expected, query execution time should not affect overall execution time significantly. Time to apply data transforms, store results in cache, and store records in the result set can also vary greatly, but since they are all dependent on the result size they can be simplified to one constant multiplier. With these assumptions, the above analysis can be simplified to:

$$N_r + N_{cr}$$

The worst-case analysis simplifies to the number of results plus the number of cached results. Since the execution time of the algorithm increases linearly with the number of results, the algorithm can be expected to perform at $O(n)$.

In the best-case scenario, the cache is up to date and there are no joins or transforms:

$$N_q * Q_c * N_{cr} * S$$

With the assumptions and simplifications described above, the algorithm is also related linearly to the number of results. In this case, the algorithm iterates through the result set only once. Since the worst and best case scenarios are both determined by the number of results, the algorithm can be described as bounded by n, or $\Theta(n)$.

It is clear from the analysis that this algorithm is not optimized for speed. It could take significant time to read and process a dataset entirely, depending on its size. The purpose of this algorithm is to integrate the decisions and tasks necessary to read from remote data sources, cache remote data if necessary, join tables from different datasets, and deliver the results as a single dataset.

Table 1. Sample results of execution time in seconds for various scenarios

Total Records	Data Operation	Without Cache	With Cache
2000	none	1.332195	0.615170
2000	none	1.304521	0.611286
2000	none	1.292177	0.620324
2000	cache update	n/a	19.537353
1014	join	19.182868	0.513660
1014	join	19.233897	0.510997
1014	join	19.273681	0.509054
2000	transformation	5.562685	0.628351
2000	transformation	5.546536	0.614257
2000	transformation	5.585319	0.614521
2000	transformation	5.560045	0.613802
2000	transformation	5.567419	0.625720
2000	transformation and cache update	n/a	23.302291

Table 1 shows a sample of execution times with and without caching using joins and transforms. This experiment shows that caching improves export speed slightly when transforms are not used. A nine hundred percent increase in speed can be observed when data transforms are used with caching. The primary purpose of data caching is to make

joins between tables from different databases possible, but performance improvements are a welcomed side effect.

The tests were performed on a dataset populated with random data. The dataset consists of two tables ("data_table" and "items_table") with the following schema:

```
data_table
  id: integer
  item_name: string
  item_value: float
  item_date: datetime

items_table
  id: integer
  item_name: string
  item_location: string
  item_family: string
```

This test dataset was created to simulate a scientific dataset. The items table contains a list of items where the item name was chosen at random from dictionary words, the location was assigned randomly from a short list of cities, and the family was also assigned randomly from a short list of possible values. A script was created to populate the items table and data table. As the script created a new item, a random number of data records were added for that item. In the data table, the item name was the same as the name from the items table, the value was a randomly generated number, and the date was randomly selected from a fixed time range. The id fields from both tables are automatically incrementing unique integers.

Generating Queries

Algorithm 3 illustrates the method used to generate the queries that read remote data sources. This function uses the export configuration to determine which tables and

fields from remote databases will be read and generates an appropriate, generic SQL query to accomplish this.

In line 1, the set of generated queries (Q) is initialized to the empty set. Line 2 populated D with all datasets being read by this export. To clarify, an export (which is itself considered a dataset) may be made up of subsets of one or more remote datasets. In order to know connection details and schema information, this function must examine the datasets accessed for this export.

Line 3 iterates through each dataset. The first step in this loop is to populate the set of tables from this data which the export will read (T) and proceed to iterate through these tables (line 5).

Line 6 assigns a string value representing the default query to the variable q. The variable t_{fields} represents a comma-separated list of fields from the table t.

If this export is configured to use caching and the cache type is incremental (line 7), the default query must be altered. Incremental caching requires that only the newest records are read from the remote data source. A numerical field (key_field) is necessary to determine the age of the data. A key_field can either be an incremental numeric id, a timestamp, or any other type of field which can be used reliably to sort records in the order in which they were added to the database. Line 8 determines if a key field has been set for the table being queried. If so, the maximum value of this key field is determined (line 9) and the query is amended to only retrieve records from the remote dataset that are newer than the locally cached records (line 10).

Line 11 adds the query generated to the set of queries and execution returns to iterating through tables and datasets if necessary.

Line 12 returns the set of all generated queries.

The execution time of this algorithm is dependent on the number of tables and number of datasets in the export. Since these numbers should not grow very large, this

function is assumed to have negligible impact on the overall execution time of the DM algorithm. The algorithm for generating queries offers a straight-forward procedure for reading all records or new records, but it is highly dependent on configuration options and use input. In an ideal representation (such as the pseudo-code presented in algorithm 3), it can be assumed that everything is configured properly. In a real-world implementation, however, a user may pick an inappropriate field as the key field or incorrectly configure the export cache. Careful error checking is necessary when implementing an algorithm so dependent upon user input.

CHAPTER V

IMPLEMENTATION DETAILS

An Apple OS X Server running Darwin Kernel 8.11.0 on a Power PC G5 CPU was used for the prototype of this system. Implementation began with the installation and configuration of PostgreSQL and Ruby on Rails on the server.

Management Interface

The web pages of the management interface were written in RHTML (HTML with Ruby code embedded in it). From this interface the user is able to manage datasets, exports, transforms, and users. Users may create, view, edit, explore, and delete datasets. When creating a dataset, the user must specify the type of database and all information necessary to connect to the database (hostname, database name, username, and password). Figure 4 shows the dataset management interface. This view allows the user to update details about the dataset and displays important information about the dataset, such as the database schema and which exports access the dataset.

Datasets may be viewed as a list from which the user can edit, delete, or explore any of the existing databases. When a database is explored, the tables are displayed with a list of fields and field types belonging to that table. By clicking on the table name the user is presented with a preview of the data. The user may further explore the dataset by typing custom SQL code. No commands that may alter the data (such as ALTER, INSERT, UPDATE, or DELETE) are supported and will return an error message if entered. Figure 5 shows the dataset exploration interface in action.

The web interface provides similar facilities for creating, viewing, editing, and deleting exports, transforms, and users. (Transforms have an interface for running the data transform on test data to verify its operation). A detailed security model was not developed

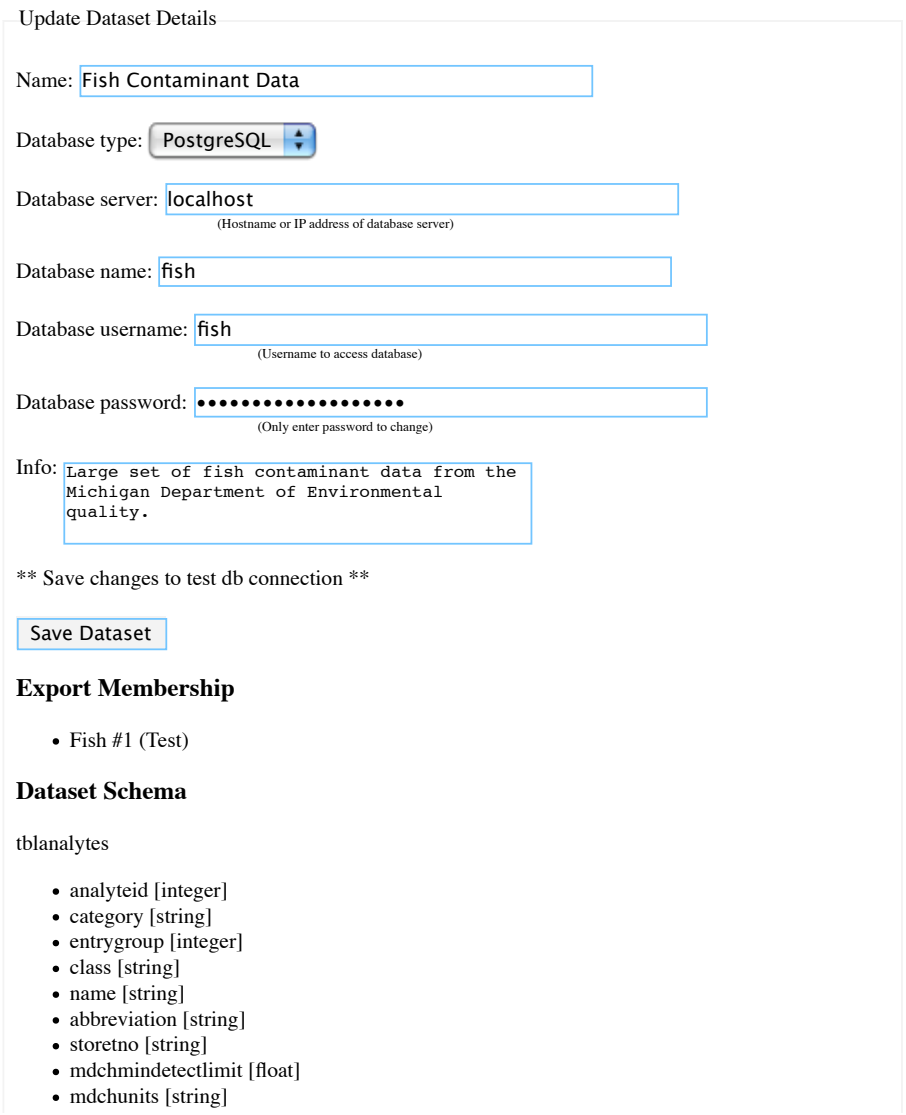

Figure 4. Dataset management interface

for this implementation so it is limited to two classes of users: guests and authorized users. Guests may view (but not create, edit, or delete) datasets, exports, and transforms. (A guest user is also unable to test data transforms). Once logged in, a user is authorized to create, edit, and delete datasets, exports, transforms, and users. Essentially, each user is an administrative user. The first user in a freshly installed system must be created manually within the database or through an installation script.

SELECT name, category, class FROM tblanalytes Go

tblanalytes

name	category	class
Mercury (ppm)	heavy metals	heavy metal
Cadmium (ppm)	heavy metals	heavy metal
Zinc (ppm)	heavy metals	heavy metal
Selenium (ppm)	heavy metals	heavy metal
Lead (ppm)	heavy metals	heavy metal
Nickel (ppm)	heavy metals	heavy metal
Chromium (ppm)	heavy metals	heavy metal
Arsenic (ppm)	heavy metals	heavy metal
Copper (ppm)	heavy metals	heavy metal
fat (%)	organics	fat
Aroclor 1242 (ppm)	organics	PCB
Aroclor 1248 (ppm)	organics	PCB
Aroclor 1254 (ppm)	organics	PCB
Aroclor 1260 (ppm)	organics	PCB
Total PCB - Aroclor (ppm)	organics	PCB
Cong06 (ppb)	organics	PCB
Cong0508 (ppb)	organics	PCB
Cong0709 (ppb)	organics	PCB
Cong1517 (ppb)	organics	PCB
Cong18 (ppb)	organics	PCB
Cong22 (ppb)	organics	PCB
Cong25 (ppb)	organics	PCB
Cong26 (ppb)	organics	PCB
Cong28 (ppb)	organics	PCB
Cong31 (ppb)	organics	PCB
Cong33 (ppb)	organics	PCB
Cong1632 (ppb)	organics	PCB
Cong37-42 (ppb)	organics	PCB
Cong40 (ppb)	organics	PCB
Cong44 (ppb)	organics	PCB

Figure 5. Dataset exploration interface

Relational Database

The database component stores all the objects managed through the web interface (datasets, exports, transforms, and users) as well as information that supports system functions and cached data from datasets. The additional objects stored in the database are joins, exported fields, and metadata fields. These objects are explained in more detail in the **Mid-**

dleware section of this chapter. Cached data is stored in dynamically created tables, which are given names that have been generated from data source queries.

Ruby on Rails uses Ruby code to define the database schema rather than allowing the user to manipulate the database schema directly. Changes to the schema may be made by adding a file containing Ruby code to the db/migrate directory inside the Rails application. When the Rails command *rake db:migrate* is executed the new file performs alterations to existing tables or creates new tables. A schema change may be rolled back in order to return the database to a previous revision.

Listing 1 shows the Ruby code used to create the datasets and metadata_fields tables. This code uses the class ActiveRecord::Schema, which is part of the Ruby on Rails installation. This and the ActiveRecord::Migration provide a structured interface by which databases used in Rails applications are managed.

The *datasets* table contains database connection information for all datasets added to the system. The *metadata_fields* table contains information about fields stored in each dataset and is updated whenever a new dataset is added or updated. This table stores the id of the dataset, a custom name for the field (may be different than the name from the original dataset), the table name, the original field name, a unique name (generated by concatenating the dataset_id, table_name, and original_name), a sample of distinct values from this field in the source database, and an indicator of whether this field is a key field. (A key field is used for incremental caching. If a field is designated as a key field it is understood that it is some sort of unique, numerical identifier in the source database. Using a key field, it should be possible to update a cache table by getting only records from the source database where the value of this field is greater than the maximum value of the corresponding cached field).

Listing 1. Sample of database schema defined in Ruby

```
create_table ‘‘datasets’’, :force => true do |t|
    t.string ‘‘name’’
    t.string ‘‘info’’
    t.string ‘‘connection_type’’
    t.string ‘‘connection_host’’
    t.string ‘‘connection_user’’
    t.string ‘‘connection_pass’’
    t.string ‘‘connection_name’’
end
create_table ‘‘metadata_fields’’, :force => true do |t|
    t.integer ‘‘dataset_id’’
    t.string ‘‘name’’
    t.string ‘‘table_name’’
    t.string ‘‘original_name’’
    t.string ‘‘field_type’’
    t.string ‘‘unique_name’’
    t.string ‘‘distinct_values’’
    t.integer‘‘key_field’’
end
```

Middleware

The middleware component provides all the necessary functionality to allow the management interface (and web service interface) to interact with the database. In Rails, the controller code is implemented in Ruby using Rails classes. The main class provided by Rails is ActiveRecord. ActiveRecord provides functionality for database connections, queries, updates, and schema manipulation. A typical database-driven web application developed with Rails would use the ActiveRecord::Base class for all database access needs. For this application, database objects created through Rails were accessed in this way. Dy-

namic database content (such as cache tables and external datasets) could not be accessed in such a simple manner.

Listing 2. Standard ActiveRecord method of reading and updating data

```
1 export = Export.find_one(id)
2 # Display export information:
3 puts export.name
4 puts export.description
5 puts export.cache_type
6 # Update cache_dirty attribute
7 export.cache_dirty = true
8 # Save export to database
9 export.save
```

Listing 2 demonstrates the use of a standard Rails database object. The code locates an export in the database which matches an id (usually supplied as a parameter to a web page) and displays some of the attributes of the export. (For this simple example error checking was omitted). The“cache_dirty” attribute is then set to true (which would trigger a cache update) and the change is saved to the database. Note the lack of any SQL in the code. Rails automatically generates the necessary SQL code but abstracts this from the programmer. This serves to simplify the process as well as make database access safer and more reliable. A drawback to this method is that the database must conform to the conventions required by Ruby on Rails.

In order to access dynamically generated tables (which do not have predefined models or schema) it is necessary to use SQL and additional database connections. ActiveRecord provides several methods to do this.

Listing 3 demonstrates how to connect to a database server and execute a query using ActiveRecord. DbAccess is a class that inherits from ActiveRecord::Base. This new

Listing 3. Manual database connection

```
1  DbAccess.clear_active_connections!
2  DbAccess.clear_reloadable_connections!
3  DbAccess.remove_connection
4  this_dataset = Dataset.find(dataset_id)
5  DbAccess.establish_connection(
6      :adapter  => this_dataset.connection_type,
7      :host     => this_dataset.connection_host,
8      :database => this_dataset.connection_name,
9      :username => this_dataset.connection_user,
10     :password => this_dataset.connection_pass
11 )
12 DbAccess.table_name = query_table_name
13 data_results = DbAccess.find_by_sql( query );
```

class works in the same way as the *Export* class in listing 2 and the *Dataset* class in listing 3 except that *DbAccess* is not associated with a database table automatically. One must manually set the table name (as shown in line 12) or Rails will assume there is a database table called“db_accesses”. Instead of reading this table as expected, DbAccess is connected to a remote database using connection information stored in the Dataset object and executes a custom query. This query is generated using the metadata associated with this dataset. The function“find_by_sql” executes the query exactly as entered. The documentation for this function stresses that it should be a“last resort” for querying a database. Since Rails performs no validation on the query, potentially harmful SQL code or database-specific code can be executed. For this implementation this “last resort” technique was necessary, since there is no way to anticipate the schema of external datasets while remaining flexible enough to allow new datasets to be added through the web interface.

Metadata Discovery

When a new dataset is added the middleware component reads table information from the source database. Table names, field names, and field types are stored in the configuration database in the metadata_fields table. The user may specify a new name or keep the original name of the field. (The original name is retained for purposes of generating queries during export). A small sample set of the values for that field are stored in the distinct_values field. This gives the user an idea about the range of data stored in the field. Listing 4 demonstrates how the application connects to a remote database and iterates through all the tables and fields and stores the metadata.

Caching

Cache tables are created dynamically and populated with processed data from remote data sources. The name of a cache table contains the id of the export and the name of the source table concatenated with other identifying information in order to ensure the table name is unique. A cache table contains only those fields from the source database table that have been chosen for export.

Web Service Interface

The web service interface utilizes the middleware and database components in much the same way as the management interface. The main differences between this view and the management interface are that the web service is completely XML and is read-only.

Figure 6 shows the export management interface. It is through this view that the user configures what fields are exported through the web service view. This view allows the user to configure caching options, add or remove fields, add or remove transformations, and add or remove joins.

Figure 7 shows the interface for adding new fields to an export. The user first chooses which dataset to select fields from and is then presented with all the fields available

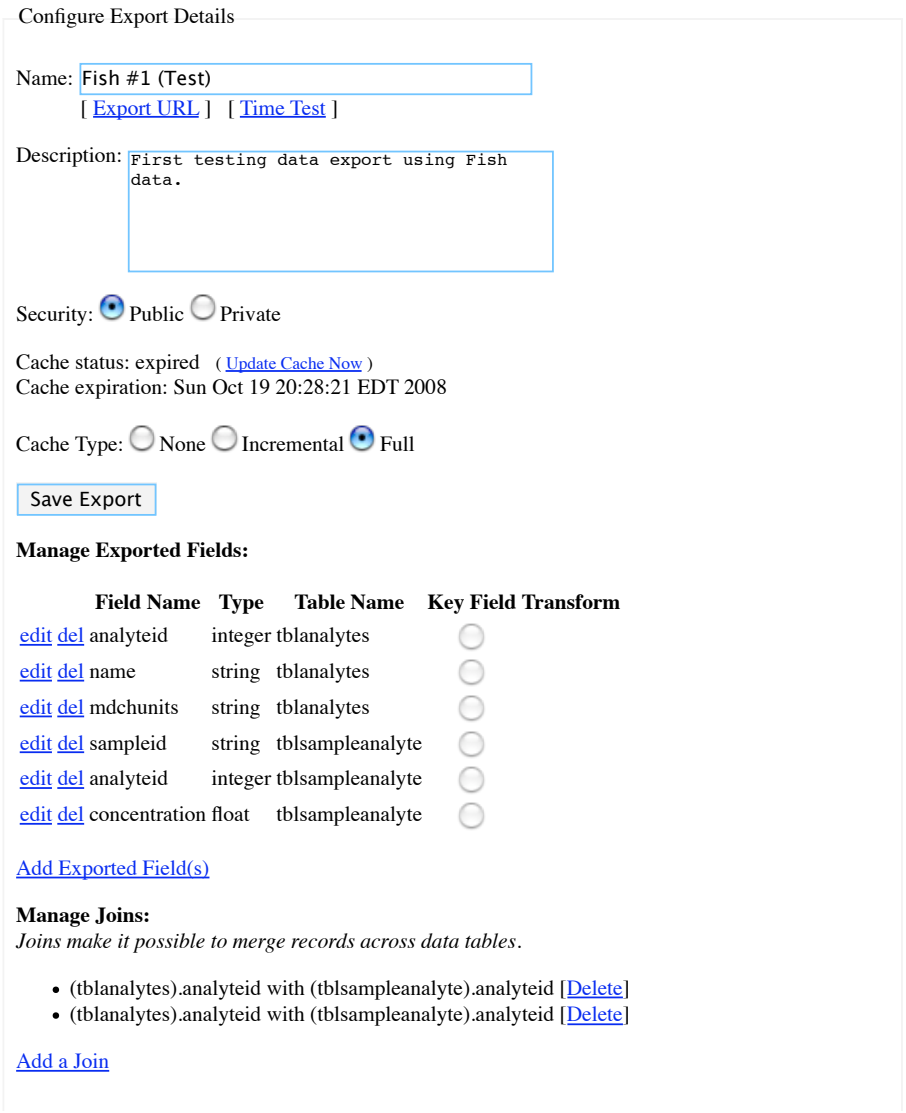

Figure 6. Export configuration interface

in that dataset. The user is shown the field name, data type, table name, and distinct values for each field.

Figure 8 shows the interface for adding new joins to an export. The user selects the tables to join and which fields to join on. The system uses an INNER JOIN, so the order of the tables affects the results of the join.

Select Dataset and Fields

Selected dataset: Fish Contaminant Data

Select field(s):

field	type	table	distinct values
☐ class	string	tblanalytes	Aldrin ,DDT ,PBB ,PCB ,Terphynyls
☐ entrygroup	integer	tblanalytes	0,1,2,3,4
☐ abbreviation	string	tblanalytes	
☐ storetno	string	tblanalytes	
☑ name	string	tblanalytes	
☐ mdchanalytes	boolean	tblanalytes	f,t
☑ mdchunits	string	tblanalytes	,ppb ,ppm
☐ mdchmindetectlimit	float	tblanalytes	0
☑ analyteid	integer	tblanalytes	1,2,3,4,5
☐ category	string	tblanalytes	heavy metals ,organics
☐ weight	float	tblfish	0,0.3,0.4,0.5,0.6
☐ length	float	tblfish	0,3.5,3.7,4,4.1
☐ visitid	string	tblfish	1998000 ,1998006 ,1998011 ,1998015 ,1998019
☐ sex	string	tblfish	,F ,M ,m
☐ age	integer	tblfish	0,1,2,3,4
☐ comment	string	tblfish	
☐ compcomment	string	tblfish	
☐ observations	string	tblfish	
☐ modifieddate	datetime	tblfish	2001-01-01 00:00:00,2001-01-02 00:00:00,2001-04-05 00:00:00,2001-05-04 00:00:00,2001-05-05 00:00:00
☐ modifiedby	string	tblfish	,BARKERJM ,BOHRJ ,Bohr ,DAYRM
			2001-01-01 00:00:00,2001-01-02

Figure 7. Export field section interface

Figure 9 shows the interface for adding new transforms to an export. The user must name the export, write a short description, select the input type (float, int, string, or nil), select the output type (float, int, string, or boolean), and then enter Ruby code to perform the actual transformation. The data is passed to the function as a variable called “input”. Once the transformation is saved it may be tested through a separate interface (see figure 10).

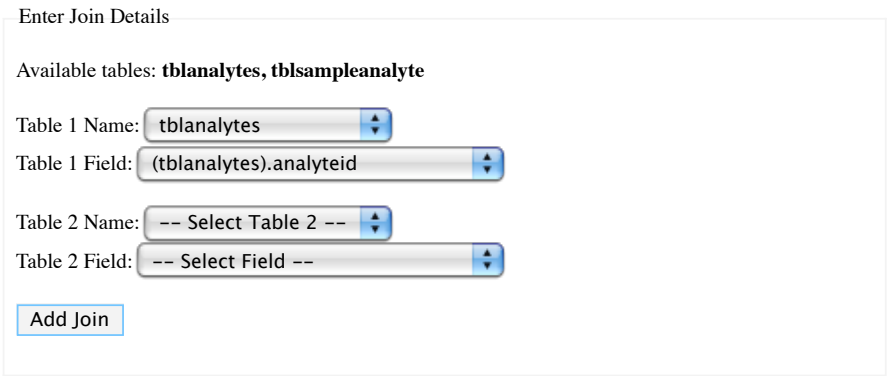

Figure 8. Join configuration interface

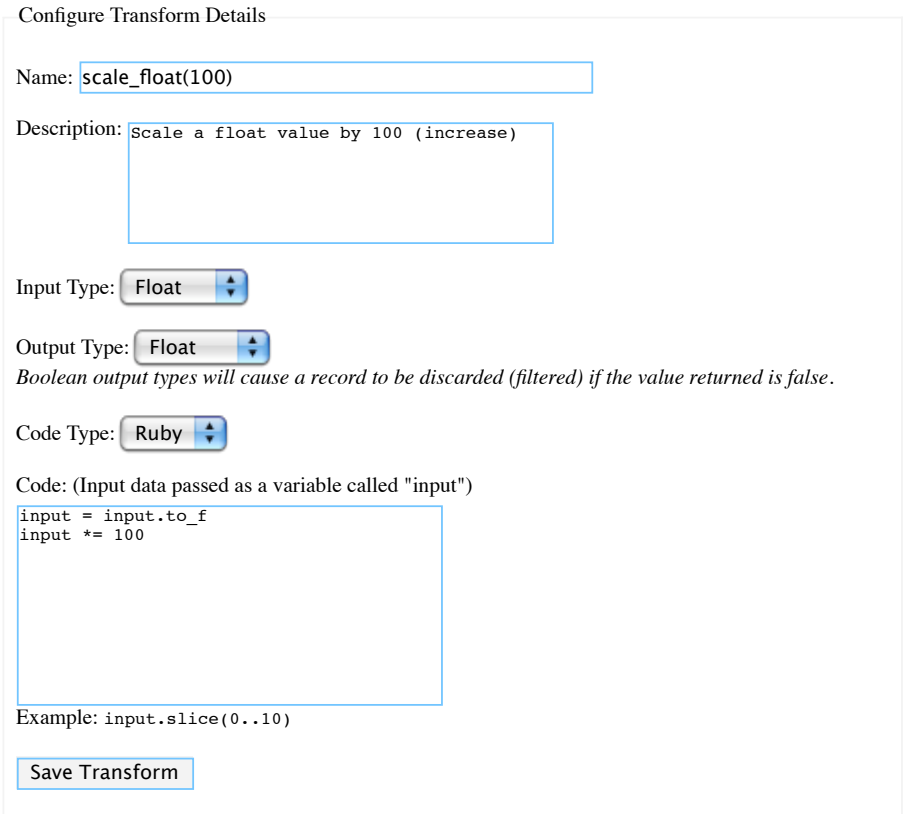

Figure 9. Transform configuration interface

Transforms can manipulate the data by scaling, adding, subtracting, or truncating the input. Transforms that return a boolean value can be used to filter or clean the source data. For example, if only data within a certain range is desired, the user can write a

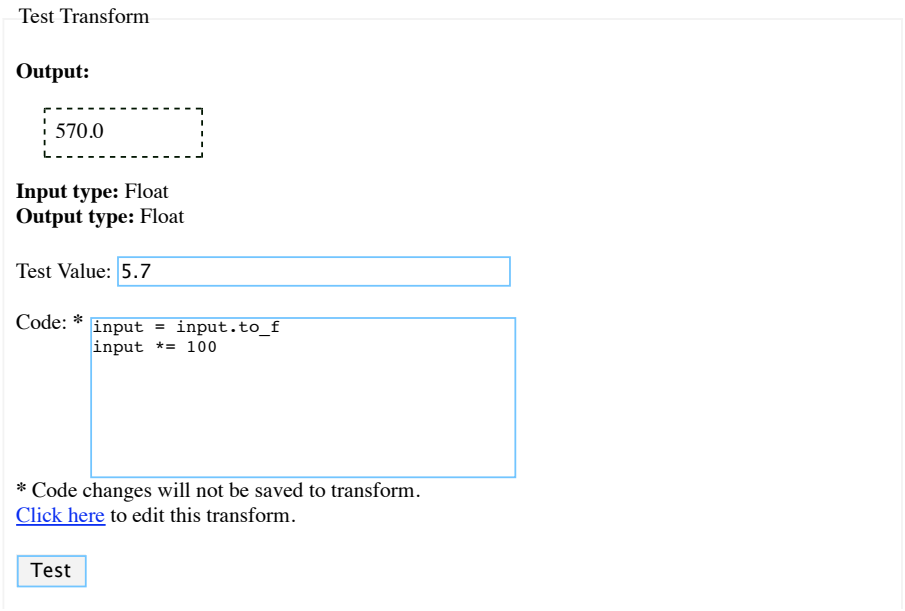

Figure 10. Transform test interface

function that returns *true* when the data falls within that range and *false* when it does not. The system will not export any records in which a transform returns a value of false.

Once exported fields, joins, and transforms are configured, the user can see the results of the export by clicking on the **Export URL** on the export interface. Data returned from a remote data source query or local cache query is in the form of an array with database columns as attributes of each item in the array. Ruby's render method converts the entire array into a properly formatted XML document as shown in listing 5.

Listing 4. Database connection and metadata discovery

```
DbAccess.establish_connection(
    :adapter  => dataset.connection_type,
    :host     => dataset.connection_host,
    :database => dataset.connection_name,
    :username => dataset.connection_user,
    :password => dataset.connection_pass
)
DbAccess.connection.tables.sort.each do |table|
    DbAccess.connection.columns(table).each do |column|
        metafield =
          MetadataField.
             find_or_initialize_by_unique_name(
            :unique_name => this_unique_name,
            :dataset_id => params[:id],
            :table_name => table,
            :original_name => column.name,
            :name => column.name,
            :field_type => column.type.to_s )
        if metafield.new_record?
            metafield.save
        end
    end
end
```

Listing 5. Example XML export

```
<?xml version="1.0" encoding="UTF-8"?>
<results type="array">
    <result>
        <analyteid type="integer">1</analyteid>
        <concentration type="NilClass">0.31</
           concentration>
        <id type="integer">1</id>
        <mdchunits></mdchunits>
        <name>Mercury (ppm)</name>
        <sampleid type="NilClass">1998000-S01</sampleid
           >
    </result>
    <result>
        <analyteid type="integer">1000</analyteid>
        <concentration type="NilClass">11.8</
           concentration>
        <id type="integer">2</id>
        <mdchunits></mdchunits>
        <name>fat (%)</name>
        <sampleid type="NilClass">1998000-S01</sampleid
           >
    </result>
</results>
```

CHAPTER VI

CONCLUSIONS

The purpose of this thesis was to design and implement a system to organize and combine datasets for analysis. The design and implementation were based on current research in data warehousing, middleware, and web applications. A set of seven criteria chosen from current literature describe the functional requirements of this system. Data caching techniques were designed and implemented in order to meet the core criteria. Also, the ability to join tables from different databases was made possible through this system. Joining tables between different databases of the same type is currently possible with proprietary extensions, but this thesis delivers the capability to join two tables from two different databases regardless of the location or type of database each table belongs to. This feature is dependent on the time necessary to extract data from remote data sources and the availability of local storage space required to cache remote data.

Provide controls and management tools to access and view data

Controls and management tools were created with Ruby on Rails and made available through a web interface. This interface allows a user to access external data sources, preview source data, configure exported data, and configure data transforms.

Support a wide variety of operating environments

When developing the prototype, tools and techniques that maximized cross-platform compatability were chosen. The areas of focus for this criterion are the server environment, database engine, and user interface.

The server application was created with Ruby on Rails. This programming environment is available for most modern operating systems, including Linux, MacOS, and

Windows. Ruby is an interpreted language, so it will run on any supported system with little or no modification to the code once Ruby on Rails is installed and configured.

The prototype system uses PostgreSQL as the configuration and caching database; however, any relational database supported by Rails may be used in place of PostgreSQL. PostgreSQL was chosen because it has been ported to most modern server platforms.

The user interface is entirely web-based and should work with any recent web browser and operating system. Only simple, standard HTML code was used to create the interface. Recent advances have brought about more desktop-like web interfaces (such as AJAX) but these technologies may behave differently depending on which browser (and which version of a particular browser) is used. The interface was tested using Ubuntu Linux, MacOS X, and Windows XP with various web browsers, including Firefox (version 2 and 3), Internet Explorer, Safari, and Google Chrome.

Support a wide range of data sources

The data sources supported by this thesis are remote databases from various vendors. The ActiveRecord class provided by Ruby on Rails supports adapters for many different relational database servers. This includes popular free and open-source databases such as PostgreSQL, MySQL, and SQLite, as well as commercial systems, such as Oracle and Microsoft SQL Server. Some database adapters are not included with Ruby on Rails by default and must be installed separately. At this time, only relational database servers are supported as data sources. File-based data formats (such as Microsoft Access files, CSV files, and XML files) are not supported directly, but a user could easily import these files into a database server in order to make the data available to the application.

Support the propagation of both incremental changes and full extracts

The caching mechanism for exports offers three caching options: no cache, incremental, and full. If no cache is selected the remote dataset is loaded each time the export

is requested. If incremental is selected only new records are loaded into cache (unless the cache somehow has been corrupted or the exported fields have been changed). Finally, a full cache type captures a snapshot of the entire source database at the time of caching. Once expired, a full cache will be loaded again in its entirety from the source database. A user may also manually clear and completely refresh the cache tables at any time, regardless of the cache type.

A user may also configure data transforms to filter exports by any criteria. In this fashion, the size and range of the output of the system can also be incremental. Ruby provides a rich set of date and time functions which would make it simple for a user to create an export limited dynamically by date or time (such as "all records from the last thirty days" or all records from a particular year).

Support source data transformations

Data transformations may truncate data, replace text, scale data, convert data to different data types, round numbers or perform any other tasks possible through Ruby code. Users can test transformations before applying to exported data. Once a transform is created it can be applied to an exported field through the export configuration interface.

A data transform is configured with an input and output type. The input types configured for this system are string, integer, float, and nil. (A nil input type may be selected when the user intends to replace the input with a fixed or random value, a timestamp, or some other output function that does not depend on the input value). The output data types are string, integer, float, and boolean.

Support data cleansing (removing inconsistent records from the data set)

Data cleansing or filtering is supported through transforms. As described in the previous section, transforms are configured with an output type. If the boolean type is selected, the transform will return true or false. Records that return false will not be exported with

the rest of the dataset. (Filtered out records will, of course, remain in the source database or cache tables). This functionality can be used to filter records of a certain age or data outside a particulate range of numbers or string constants.

Support documented SQL and other open interfaces for third-party and user-defined code

Browsing the data available within a dataset is possible using standard SQL. A user may explore the dataset using SQL SELECT statements but no SQL statements that could potentially modify the source database will be processed. Disallowed functions include INSERT, DELETE, UPDATE, DROP, and ALTER. If the user attempts to use any of these SQL commands (either in the main query or a sub-query), an error message will be displayed.

The data transform interface also meets this criterion by allowing the use of standard Ruby code.

Data Caching

While data caching was not part of the original seven criteria, it became a necessary part of the prototype system. Data caching is accomplished by replicating subsets of remote data sources within the configuration database. This serves to improve response time of exports (particularly when data transforms are involved) and make it possible to join tables from different datasets.

Remote Joins

A feature of this application is the ability to join tables from different databases. While some database engines support joining tables from two different servers running the same database engine, research has turned up no products that support joining tables from different database engines. The application designed for this thesis can be configured to

join data from any two datasets that are configured in the system even if these datasets are on different servers and are different database engines.

Remote joins are accomplished by caching fields from external data sources and joining the locally cached tables. If an export has not been configured to use caching, a temporary set of cache tables is created, joined, and then discarded after the results are exported. Exports configured to use caching will respond much more quickly (unless the cache tables need to be refreshed).

Future Work

While this thesis uses metadata and allows the user to view metadata information (such as table structures and distinct values), no facility for exporting metadata has been included.

This system imports data from relational databases only. While a wide variety of database systems are supported, text-based data formats, such as Comma Separated Value (CSV) and XML are not supported. Other common binary formats (such as Microsoft Access) are also unsupported. These unsupported formats could be converted into a relational database (Microsoft SQL Server provides a simple tool for importing databases), but this additional step may not be possible for all users due to lack of resources of technical expertise.

In addition to expanding the range of data source types, it would be helpful to expand the range of output types. Currently the only export format is XML. Export data in CSV, SQL, and other formats may simplify the analysis process and give the user more options when selecting data analysis tools.

Data transforms can only be written in Ruby. Transforms written in SQL may improve performance by performing transformations on an entire result set at the database level, as opposed to iterating through a result set and performing the transformation on

each record. The difficulty with using SQL is that not all database engines support the exact same version of SQL. Introducing SQL code also makes it possible for a user to inadvertently (or purposely) alter the source dataset.

Security was not a high priority when designing this system. The simple user hierarchy implemented adequately prevents unauthorized users from changing dataset, export, and transform information, but it is not sufficiently fine-grained enough to determine which users have permission to change which objects. A more complex security system with access control lists would be helpful, if not essential, if this application were to be accessed by several users.

Exports are configured by picking fields from external datasets and applying transforms or joins if required. Advanced users may find it easier and more flexible to configure exports using one or more custom SQL queries. Such queries could pick database fields and tables, perform data transformation and filtering, and even perform joins on tables within the same dataset before bringing the data into the cache tables.

The system currently runs on a single server and has no mechanism for distributed, grid-based, or agent-based operation. By distributing the workload over several servers or across a greater area, it would be possible to improve the speed, efficiency, and capacity of this system.

Recommendations

The benefits of sharing scientific data between and within organizations cannot be denied. Many tools exist to make this possible but these tools often require significant technical expertise to implement. More user-friendly commercial tools exist, but these come at a higher price and often require additional yearly license fees.

Tools such as the one described in this thesis can offer a user-friendly, inexpensive method of sharing data. It is recommended that researchers involved in collecting

large amounts of data consult with their information technology departments and setup a proper database engine for structured, long-term, reliable storage of collected information. College or university computer science departments may be able to offer assistance if a research organization does not have the information technology resources to install or manage a database server.

APPENDICES

APPENDIX A

MANUAL

This manual describes the features of the DM system version **1.0**. It is assumed that the user has a properly installed and configured system already. All information about installation and configuration are found in the Installation Guide. This user's manual provides a brief overview of the system and detailed instructions on using each of the main features. Advanced features are also covered.

Disclaimer

This software is available free of charge for any purpose, including (but not limited to) educational and commercial. This software is provided on an "as is" basis without warranty of any kind, express or implied.

Overview

The DM system allows a user to connect to one or more remote databases and export, combine, and explore the information contained within the data sources. DM does not stand for anything in particular but it originally meant "datamining." The main algorithm implemented in this system became known as the "dynamic-merged caching algorithm" or DM Algorithm.

This system was created by Wesley Leonard as part of a thesis project as part of the requirements for a master's degree in computer science from Central Michigan University.

Technical Details

The DM system was written in Ruby on Rails. Any relational database supported by Ruby on Rails may be used as the main configuration and caching database. PostgreSQL was used while developing this system and is the preferred database engine. Changes to the

database connection may be made by editing the “database.yml” configuration file found in the “dm_middleware/config” directory. See the Installation Guide or any reference that covers Ruby on Rails database configuration for more details.

User Interface

The user interface is entirely web-based and should work with any modern browser. The interface is divided into four main sections: Datasets, Exports, Transforms and Users. Guest (or unauthenticated) users may view available datasets and exports, but only authenticated users may perform any management tasks such as adding, editing, or deleting objects.

If the system is being accessed for the first time, a user must login with the default username and password. These values are “setup” and “fishfood”, respectively. After initial login, it is up to the user to create a new user and delete the default user. (Leaving the default user active may result in the system being accessed by unauthorized users).

Datasets

Datasets are the sources of data for the system. Currently, each dataset must be a relational database engine that is supported by Ruby on Rails. Future releases may support file-based datasets (such as CSV or XML files) or other methods of importing data.

Clicking on the datasets tab displays a list of all existing datasets and a truncated descriptions of each dataset. The user can also click to add a new dataset or delete/edit/explore existing datasets.

Add

When adding a new dataset, the user specifies the dataset name (which must be unique) and a short description of the dataset. The following database connection details are also required: database type (chosen from a list of supported databases), database server

(hostname or IP address), database name, database username, and database password. If the dataset was successfully added the user will be returned to the list of datasets.

Table 2. Supported database engines in Ruby on Rails

Supported Databases
MySQL
PostgreSQL
SQLite
SQL Server
IBM DB2
Informix
Oracle
Firebird/Interbase LDAP (ActiveLDAP)
SybaseASA (Sybase Adaptive Server Anywhere or SQL Anywhere Studio)
MonetDB (see http://rubyforge.org/projects/monetdb-ror)

Table 2 shows which database engines are currently supported by Ruby on Rails and, by extension, this application. Table 3 lists database engines that are currently being implemented. See: http://wiki.rubyonrails.org/rails/pages/DatabaseDrivers for the most up-to-date list of supported databases and databases in development.

Table 3. Database engines under development

Database	Status
Cach	Object interface almost implemented (SQL interface works)
OpenBase	Works on OS X, Linux and Windows under development
SQLAnywhere	C implementation (previously Sybase iAnywhere's ASA)

Delete

A user may click `del` to completely remove any of the datasets listed. A prompt is presented to confirm that the user really wants to delete the dataset. The user may click `cancel` to keep the dataset or `ok` to confirm deletion.

Edit

By clicking `edit` the user is taken to the dataset editing screen where dataset details, such as the name, description, and database connection information can be updated. This screen also shows the export membership (which exports are configured to use this dataset) and the database schema (tables, field names, and field types) of the dataset.

Explore

The user may preview the data contained in the source dataset by clicking `explore`. Initially, the user is presented with the database schema where each table is a clickable link. When the table name is clicked, the first fifty rows of data are presented to the user. The user may replace the default SQL statement (`SELECT * FROM` *`TABLE_NAME LIMIT`* `50`) with any other valid SQL statement. Through this interface the user can explore the entire database table, select a subset of fields from the table, experiment with joins, determine distinct values, and get acquainted with the data. No commands which could change the source database in any way are permitted. These forbidden commands include `DELETE`, `INSERT`, `UPDATE`, `ALTER`, and `DROP`.

Exports

Exports are subsets or combinations of data from source databases presented in XML format. An export may apply transformations and joins to records from one or more datasets.

Add

A name and description are all that is required to add an export. After adding a new export the user is taken to the edit screen to continue configuration of the new export. The export name must be unique.

Delete

From the exports list, a user may click `del` to completely remove an export. A prompt is presented to confirm that the user really wants to delete the export. The user may click `cancel` to keep the export or `ok` to confirm deletion.

Edit

The edit screen is shown immediately after adding a new export or when `edit` is clicked on the exports list. From this screen the user can update the name and description of the export as well as the cache type. Cache type choices include none (caching is not used), incremental (new cache data is appended to existing cache data when refreshed), and full (the entire cache is refreshed upon expiration). The edit screen also displays the cache status for exports configured to use caching. Cache information can be clean (cache is up-to-date), expired (cache has not been updated since the expiration time), or dirty (the export configuration has changed and a cache update is required). A user may force a cache refresh at any time by clicking `Update Cache Now`.

The list of fields included in the export is shown under the **Manage Exported Fields** heading. Exported fields may be added, edited, or deleted. The field name, data type, table name, key field indicator, and data transforms (if applied) are listed for each exported field. When adding a field, the user first selects the source dataset and then chooses fields from this dataset for inclusion. When selecting a field, the field name, data type, table name, and a small set of distinct values are shown. Distinct values give the user an idea of what type of data is in the field and what constraints, if any, are placed on the data. For example, distinct values may be limited to “yes” or “no”, a short list of string values, or a small set of numbers. If incremental caching is used, the Key Field indicator can be selected for only one field from each remote database table. Designing a field as the key field means that it is used to determine the order in which records are added to the remote

dataset. With incremental caching, only records added after the locally cached records are extracted from the remote data source. After adding a field a user may edit the field to apply a data transform (the next section covers transforms in detail).

The edit export screen also supports the addition and deletion of joins. A join is a database operation that brings together data from different tables based upon a common piece of data (called a join predicate). Often joins are performed using a numeric identifier as the join predicate but any type of data can be used to join tables. Adding a join to an export requires selecting two tables and the join field from each table. A common reason for joining tables is to build a more complete set of records for inspection or analysis. For example, a database may have a table that stores employee information (including their employee id number, name, and office number). Another table may contain expenses charges to an employee. This table would likely contain the employee id, amount of expense, date, and description of the expense. These two tables could be joined using the employee id as the predicate in order to generate a convenient list of expenses for one or more employees which includes the employee name and contact information.

Export URL

The `Export URL` link provides access to the actual exported data in XML format. This link is found both on the list of exports and on the edit forms for each individual export. This URL may be accessed through a standard web browser to preview the data, downloaded to an XML file for analysis, or accessed by an analysis tool directly.

Transforms

Transforms allow the user to scale, filter, or simplify data for analysis. A good example of when a scaling transform is useful is when data from two different datasets is combined. One dataset may have used percentages (0 - 100) while another may have used decimals (0.0 - 1.0). In this case the user could configure the DM system to scale down

the percentages by 100 or scale up the decimal values by 100 so that the datasets could be combined and analyzed with the same set of tools.

Add

Adding a transform requires a name, description, input data type (string, float, integer, or nil), output data type (string, integer, float, boolean), and a block of Ruby code. The Ruby code provided is executed each time the export is viewed (or cached). For string, integer, and float output types, the result of the code provided simply replaces the original data value. If boolean is chosen as the output type, then a row of data will be skipped if the transform code evaluates to false. In this way the user can set up constraints to ensure the data falls within certain limits or matches certain criteria.

The value read from the database is passed into the code as a variable called `input`. Here is an example of the code used to scale input by 100:

```
input = input.to_f
input *= 100
```

The first line verifies that the input is of type float by explicitly converting it. The second line multiplies the input by 100. The value returned by the last line of the code is what is returned.

Here is another example using a string input type:

```
input = input.slice(0..10)
```

This code truncates a string to ten characters.

Delete

A user may click `del` to completely remove any of the transforms listed. A prompt is presented to confirm that the user really wants to delete the transform. The user may click `cancel` to keep the transform or `ok` to confirm deletion.

Edit

Editing a transform allows the user to change all fields including name, description, input type, output type, and code. After adding or editing it is recommended that the transform be tested to verify it works correctly.

Test

The transform test screen shows input and output type as well as the code for the transform. A value can be placed in the test value field and the transform code will treat that as if it was data read from a database. After clicking `Test` result is displayed.

Users

Users are authorized to login and perform all tasks available. In the current version of the DM system there is no concept of security or differentiation between users.

Add

A new user may be added by supplying a name and password. This user will then have access to all aspects of the application, including the ability to add and delete other users.

Delete

A user may click `del` to completely remove any of the users listed. A prompt is presented to confirm that the user really wants to delete the user. The user may click `cancel` to keep the user or `ok` to confirm deletion.

Edit

Clicking on `edit` will allow a user's name or password to be changed.

Advanced Features

The features discussed in this section are built into the application but are not available through the standard user interface. These features may require the user to append information to the standard URL or edit configuration files. Caution should be taken when using any advance feature and backups of configuration files should be made prior to making any changes.

Force Cache Update

Usually a cache update occurs after a certain length of time (the default for this application is seven days) or when the export has been changed by adding or removing fields, joins, or transforms. In some cases the user may wish to force a cache update. In this situation, the user can append **?cache_refresh=true** to the export URL. (An export URL will be in the format *http://SERVERNAME:PORT/webservice/show_export/ID* where *SERVERNAME* is the name or IP address of the server hosting the application, *PORT* is the port number on which the application is running, and *ID* is the export id number).

Speed Test and Export Details

A user may wish to learn more about the export without actually viewing the exported data. In this situation, the user can append **?time_test=true** to the export URL. This will cause the web service to execute all steps necessary to export the dataset but will show information about the dataset in a standard web page instead of showing the data in XML format. This feature is designed primarily for speed testing various export configurations. The information returned includes the time necessary to process the export, the export name, the number of records exported, the number of joins used, whether caching was used, whether the cache was updated, and the current cache expiration time.

Changing the Appearance of the Application

The appearance of the system is based on cascading style sheets (CSS). From within the root directory of the application, the main style sheet is **public/stylesheets/standard.css**. A user with sufficient access to the server environment and privileges may edit this file.

APPENDIX B

INSTALLATION GUIDE

This guide describes the requirements and actions necessary to install and configure the DM system on a compatible server or personal computer.

Requirements

The DM system requires Ruby on Rails and a relational database server. These may be installed on the same computer or two separate computers. The DM system has been tested with the following configuration:

- MacOS X Server with Darwin Kernel 8.11.0, Power PC G5 CPU
- Ruby 1.8.2
- Rails 2.02
- PostgreSQL 8.2.4

Any operating system and hardware platform that supports Ruby on Rails should be able to run the DM system. Software versions newer than those listed above should also work. Ruby on Rails can be obtained from `http://www.rubyonrails.org`. This web site also provides extensive installation information and tutorials.

The DM system was developed with PostgreSQL (`http://www.postgresql.org`) as the configuration database but should work with any database supported by Ruby on Rails. See the DM User Manual or the Ruby on Rails web site for information on which databases are currently supported.

Installation

The DM system files may be downloaded in one of three formats:

- Compressed tar archive (.tar.gz) – Appropriate for Unix-like systems such as Linux and Apple OS X server. Copy the distribution file into the desired directory and execute the command: tar -xzf filename.tar.gz
- Winzip file (.zip) – Appropriate for Windows or Unix-like systems. Copy this file into the desired directory and extract it.
- VMWare image – A complete Ubuntu Server operating system with Ruby on Rails and the application already installed. Simply load this into any compatible VMware player and start.

If installing from an archive file (.tar.gz or .zip), copy this file into any directory on the server and extract it. After extraction, a directory called "dm_middleware" should appear. This directory contains all Ruby code and configuration files for the system. This directory may be moved anywhere on the server.

If installing from a VMWare image, simply load the image using VMWare player or ESX server. An Ubuntu Linux server will boot up and the DM system will launch automatically. The DM system will appear as a web site on port 8082 of the running VMWare instance. The network on the system is configured to use DHCP. Log in with the username "root" and the password "fishfood" to change network and system configurations. This is the administrative account and it can only be accessed from the console. Network logins for this user have been disabled. It is still highly recommended that you create a regular user and change the root user password immediately. For more information on Linux system administration see *The Linux System Administrator's Guide* at `http://www.tldp.org/LDP/sag`.

The VMWare image also provides a fully configured PostgreSQL server with the configuration database already installed. Since there is no need to configure the database with the VMWare version, the following section may be skipped.

Configuration

After extracting the system files into the desired directory, the configuration database needs to be created on the database server. A new or existing user must have permission to create, alter, and drop tables in the database. Also, the user should have permission to select, insert, update, and delete records within tables in this database. Configuration options vary between different database types. For most databases, the user will simply be given the role of "database owner" on the configuration database. The database may be given any name.

To configure the system to use the configuration database, change into the dm_middleware directory and open the file `config/database.yml`. Under the **production:** section, enter the database connection details as follows:

```
production:
  adapter:  DATABASE_TYPE
  database: DATABASE_NAME
  username: USERNAME
  password: PASSWORD
  host: HOSTNAME_OR_IP
```

In the example above, adapter refers to the type of database used. Consult the Ruby on Rails web site for details. Common types include `mysql`, `postgresql`, and `oracle`. Database, username, and password refer to the name of the configuration database, the username needed to access the database, and the password, respectively. Finally, host is

the hostname or IP address of the database server. Use “localhost” if the database engine is running on the same server as the DM system.

Starting the Application

The DM system is launched like any standard Ruby on Rails application. To launch an instance of the application, change to the dm_middleware directory and execute the following command:

```
ruby ./scripts/server -p 8020
```

In the above command, `ruby` refers to the Ruby interpreter. If the Ruby interpreter is not in the path of the user starting the system the full path to the Ruby executable may need to be supplied. The final argument (`8020`) is the TCP port number. Any port number may be substituted as long as the user has permission to open that port. On a Unix-like system, the ampersand (&) may be added to the end of the launch command to put the process in the background.

The method of starting processes at boot time differs between operating systems. On a Unix-like system, the launch command may be placed in a script file that is executed at boot time. In this situation, the full path to the Ruby interpreter and the “server” script will need to be supplied. It is highly recommended that the command be executed by a user with no administrative privileges.

Once started, the application will be available on the selected port. One can access the application through a web browser. If a firewall is used on the system on which the DM application is installed, be sure to open that port if other computers will need to access the application.

APPENDIX C

SELECTED CODE

The following code makes up the controller portion of the application. All algorithms presented in this thesis are implemented in this Ruby code.

Listing 6. Login Controller

```
1  class LoginController < ApplicationController
2    #
3    # Functions related to creating, editing, and
       authenticating users.
4    #
5
6    layout "standard"
7
8    #
9    # Create a new user:
10   #
11   def add_user
12     @user = User.new(params[:user])
13     if request.post? and @user.save
14       flash.now[:notice] = "User <b>#{@user.name}</b>
           created"
15       @user = User.new
16     end
17   end
18
19   #
20   # Edit user information:
21   #
22   def edit_user
23     @user = User.find(params[:id])
24     if request.post?
```

```
25
26      @user = User.update(params[:id], params[:user]);
27
28      if @user.save
29        flash.now[:notice] = "User <b>#{@user.name}</b>
              updated"
30      else
31        flash.now[:notice] = "Error updating user"
32      end
33    end
34  end
35
36  #
37  # Process user login:
38  #
39  def login
40    session[:user_id] = nil
41    if request.post?
42      user = User.authenticate(params[:name], params[:
           password])
43      if user
44        session[:user_id] = user.id
45        session[:user_name] = user.name
46        uri = session[:original_uri]
47        session[:original_uri] = nil
48        redirect_to(uri || { :action => "index" })
49      else
50        flash[:notice] = "Login failed"
51      end
52    end
53  end
54
55  #
56  # Process user logout:
57  #
```

```
58   def logout
59     session[:user_id] = nil
60     session[:user_name] = nil
61     flash[:notice] = "Logged out"
62     redirect_to(:action => "login")
63   end
64
65   #
66   # Delete a user:
67   #
68   def delete_user
69     if request.post?
70       user = User.find(params[:id])
71       user.destroy
72     end
73     redirect_to(:action => :list_users)
74   end
75
76   #
77   # Query all users:
78   #
79   def list_users
80     @all_users = User.find(:all)
81   end
82
83   #
84   # Authenticate user login information:
85   #
86   def authorize
87     unless User.find_by_id(session[:user_id])
88       session[:original_uri] = request.request_uri
89       flash[:notice] = "Please log in"
90       redirect_to(:controller => "login", :action => "
           login")
91     end
```

```
  end

end
### End login_controller ###
```

Listing 7. Manage Controller

```
###
### Collection of classes and methods for managing datasets
###

#
# Abstract classed used for ad-hoc database access:
#
class DbAccess < ActiveRecord::Base
  self.abstract_class = true
end

#
# A second abstract classed used for ad-hoc database access:
#
class DbAccess2 < ActiveRecord::Base
  self.abstract_class = true
end

class ManageController < ApplicationController
  #
  # Functions related to creating, editing, and managing datasets and metadata
  #

  layout "standard"

  #
```

```
27  # List of supported database types (a subset of types
        supported by Rails):
28  #
29  def get_db_types
30    { "PostgreSQL" => "postgresql", "MySQL" => "mysql",
          "Oracle" => "oracle" }
31  end
32
33  #
34  # Create a new dataset:
35  #
36  def add_dataset
37    @db_types = get_db_types
38    @dataset = Dataset.new(params[:dataset])
39    if request.post? and @dataset.save
40      flash.now[:notice] = "Dataset <b>#{@dataset.name
            }</b> created"
41      redirect_to( :action => "list_datasets" )
42    end
43  end
44
45  #
46  # Delete a dataset:
47  #
48  def delete_dataset
49    if request.post?
50      dataset = Dataset.find(params[:id])
51      dataset.destroy
52    end
53    redirect_to( :action => "list_datasets" )
54  end
55
56  #
57  # View a dataset through SQL queries:
58  #
```

```
59  def explore_dataset
60    @dataset = Dataset.find(params[:id])
61    @metadata_fields = MetadataField.find(:all, :
         conditions => [ "dataset_id=?", params[:id] ], :
         order => "table_name ASC" )
62    @metadata_tables = MetadataField.find(:all, :
         conditions => [ "dataset_id=?", params[:id] ], :
         select => "DISTINCT table_name" )
63
64    if params[:table_name]
65
66        # Ad-hoc access to selected data source:
67        DbAccess.clear_active_connections!
68        DbAccess.clear_reloadable_connections!
69        DbAccess.remove_connection
70
71        DbAccess.establish_connection(
72            :adapter  => @dataset.connection_type,
73            :host     => @dataset.connection_host,
74            :database => @dataset.connection_name,
75            :username => @dataset.connection_user,
76            :password => @dataset.connection_pass
77        )
78
79        DbAccess.table_name = params[:table_name]
80
81        @sql_statement = "SELECT * FROM " + params[:
             table_name]+ " LIMIT 50"
82        @error_msg = ""
83
84        if params[:sql_statement]
85          # make sure no "UPDATE" or "DELETE" or other
               statements used:
86          if params[:sql_statement].match(Regexp.new("
               delete from",true))
```

```
87            @error_msg += "Cannot delete!!!"
88          elsif params[:sql_statement].match(Regexp.new
               ("insert",true))
89            @error_msg += "Cannot insert!!!"
90          elsif params[:sql_statement].match(Regexp.new
               ("update ",true))
91            @error_msg += "Cannot update!!!"
92          elsif params[:sql_statement].match(Regexp.new
               ("alter ",true))
93            @error_msg += "Cannot alter!!!"
94          elsif params[:sql_statement].match(Regexp.new
               ("drop ",true))
95            @error_msg += "Cannot drop!!!"
96          else
97            @sql_statement = params[:sql_statement]
98          end
99        end
100
101       begin
102           @table_data = DbAccess.find_by_sql(
                 @sql_statement )
103       rescue
104           @sql_statement = "SELECT * FROM " + params
                 [:table_name]+ " LIMIT 51"
105           @table_data = DbAccess.find_by_sql(
                 @sql_statement )
106       end
107     end
108
109   end
110
111   #
112   # Edit a dataset:
113   #
114   def edit_dataset
```

```
115     @db_types = get_db_types
116     @dataset = Dataset.find(params[:id])
117     if request.post?
118
119       @dataset = Dataset.update(params[:id], params[:
             dataset]);
120
121       if @dataset.save
122         flash.now[:notice] = "Dataset <b>#{@dataset.
               name}</b> updated"
123       else
124         flash.now[:notice] = "Error updating dataset"
125       end
126     end
127
128     # Scan dataset to update metadata and related
          information:
129     @export_display = "     <i>This dataset
          is not included in any exports at this time.</i>
          "
130     @schema_display = ""
131
132     field_ids  = Array.new
133     export_ids = Array.new
134
135     metadata_fields = MetadataField.find(:all, :
          conditions => [ "dataset_id=?", params[:id] ] )
136     metadata_fields.each do |field|
137       field_ids.push field.id
138     end
139     if field_ids.length > 0
140       exported_fields = ExportedField.find(:all, :
            conditions => { :field_id => field_ids } )
141       exported_fields.each do |field|
142           export_ids.push field.export_id
```

```
143       end
144       exports = Export.find( export_ids );
145       @export_display = "<ul>"
146       exports.each do |export|
147         @export_display += "  <li>" + export.name + "</
              li>"
148       end
149       @export_display += "</ul>"
150     end
151
152     DbAccess.clear_active_connections!
153     DbAccess.clear_reloadable_connections!
154
155     DbAccess2.clear_active_connections!
156     DbAccess2.clear_reloadable_connections!
157
158     if not @dataset.connection_type.empty?
159       begin
160         DbAccess.establish_connection(
161           :adapter  => @dataset.connection_type,
162           :host     => @dataset.connection_host,
163           :database => @dataset.connection_name,
164           :username => @dataset.connection_user,
165           :password => @dataset.connection_pass
166         )
167
168         DbAccess2.establish_connection(
169           :adapter  => @dataset.connection_type,
170           :host     => @dataset.connection_host,
171           :database => @dataset.connection_name,
172           :username => @dataset.connection_user,
173           :password => @dataset.connection_pass
174         )
175
176         DbAccess.connection.tables.sort.each do |table|
```

```
177             @schema_display += table + "<ul>"
178             DbAccess.connection.columns(table).each do |
                   column|
179
180               # Read schema information from remote data
                     source:
181               this_unique_name = params[:id] + "-" +
                     table + "-" + column.name
182               metafield = MetadataField.
                     find_or_initialize_by_unique_name( :
                     unique_name => this_unique_name, :
                     dataset_id => params[:id], :table_name
                     => table, :original_name => column.name,
                      :name => column.name, :field_type =>
                     column.type.to_s )
183
184               @schema_display += "<li>" + column.name
185               @schema_display += " [" + column.type.to_s
                     + "] "
186
187               if metafield.new_record?
188                 metafield.save
189               end
190
191               this_distinct_values = ""
192
193               begin
194               # Populate distinct_values field:
195                 if metafield.distinct_values.to_s.empty?
196                   DbAccess2.set_table_name(table)
197                   DbAccess2.reset_column_information
198                   distinct_sql = "SELECT DISTINCT " +
                        column.name + " AS distinct_val FROM
                         " + table + " LIMIT 5"
```

```
199                 distinct_results = DbAccess2.
                       find_by_sql( distinct_sql )
200                 if distinct_results != nil and
                       distinct_results.size > 0
201                   distinct_results.each do |dv|
202                     this_distinct_values += dv.
                           attributes["distinct_val"] + ","
203                   end
204                   metafield.distinct_values =
                         this_distinct_values.sub(/,$/,"")
205                   metafield.save
206                 end
207               end
208             rescue
209             end
210           end
211           @schema_display += "</ul>"
212         end
213       rescue
214         flash.now[:notice] = "Unable to connect to
              database."
215       end
216     end
217
218   end
219
220   #
221   # Generate list of all datasets:
222   #
223   def list_datasets
224     @all_datasets = Dataset.find(:all)
225   end
226
227 end
228 ### End manage_controller ###
```

Listing 8. Webservice Controller

```
###
### Collection of classes and methods for and exporting
    and transforming data
###

#
# Abstract classed used for ad-hoc database access:
#
class DbAccess < ActiveRecord::Base
  self.abstract_class = true
end

#
# A second abstract classed used for ad-hoc database
   access:
#
class Result < ActiveRecord::Base
  self.abstract_class = true
end

class WebserviceController < ApplicationController
  #
  # Configure exports and exported fields
  #

  layout "standard"

  #
  # Create a new export:
  #
  def add_export
    @export = Export.new(params[:export])
    @export.cache_dirty = true
    @export.cache_expiration = Time.now
```

```
33      if request.post? and @export.save
34        flash.now[:notice] = "Export <b>#{@export.name}</
            b> created"
35        redirect_to( { :action => "edit_export", :id =>
            @export.id } )
36      end
37    end
38
39    #
40    # Remove an export:
41    #
42    def delete_export
43      if request.post?
44        export = Export.find(params[:id])
45        export.destroy
46      end
47      redirect_to(:action => :list_exports)
48    end
49
50    #
51    # Update an export:
52    #
53    def edit_export
54      @export = Export.find(params[:id])
55      @export.cache_type = @export.cache_type.to_i
56
57      export_id = params[:id]
58      @exported_fields = ExportedField.find(:all, :
            conditions => [ "export_id=?",export_id ], :
            order => "field_id" )
59      @existing_joins = ExportJoin.find(:all, :conditions
            => [ "export_id=?",export_id ] )
60
61      if request.post?
```

```
62      @export = Export.update(params[:id], params[:
           export])
63      @export.cache_dirty = true
64      if @export.save
65        flash.now[:notice] = "Export <b>#{@export.name
             }</b> updated"
66      else
67        flash.now[:error] = "Error updating export"
68      end
69    end
70
71  end
72
73  #
74  # List all exports:
75  #
76  def list_exports
77    @all_exports = Export.find(:all)
78  end
79
80  #
81  # Display an export in xml (implementation of DM
       algorithm):
82  #
83  def show_export
84
85    @start_time = Time.now
86    ###
87    ### First, get info about the export being shown:
88    ###
89    this_export = Export.find(params[:id] )
90
91    ###
92    ### Get info about exported fields:
93    ###
```

```
94      fields = ExportedField.find(:all, :conditions => [
           "export_id=?",params[:id]] )
95
96      ###
97      ### Load joins:
98      ###
99      joins = ExportJoin.find( :all, :conditions => ["
           export_id=?",params[:id] ] )
100
101     ###
102     ### Store list of ids of metadata fields:
103     ###
104     ids = fields.map { |field| field.field_id }
105
106     ###
107     ### Get info from metadata fields:
108     ###
109     these_fields = MetadataField.find(ids, :order => "
           table_name" )
110
111     ###
112     ### Build quer(y|ies):
113     ###
114     @debug_msg = ""
115     dataset_ids = Array.new
116     queries = Array.new
117     query_fields = Array.new
118     query_table = ""
119     cache_tables = Array.new
120     new_data_results = Array.new
121     data_transforms = Hash.new
122
123     transformed_fields = ExportedField.find(:all, :
           conditions => [ "transform>0 AND export_id=?",
           params[:id] ] )
```

```
124     transformed_fields.each do |trans_field|
125       tmp_field = MetadataField.find(trans_field.
             field_id)
126       data_transforms[tmp_field.table_name+"."+
             tmp_field.name] = trans_field.transform
127     end
128
129     # Loop through fields, organized by table name, and
          build queries:
130     these_fields.each do |tf|
131       # Initialize query_table:
132       if query_table == ""
133         query_table = tf.table_name
134         dataset_ids.push tf.dataset_id
135       end
136       if query_table != tf.table_name
137         # Finish query, push onto list of queries:
138         queries.push "SELECT " + query_fields.join(",")
              + " FROM " + query_table
139         cache_tables.push "cache_table_" + params[:id]
             + "_" + query_table
140         # Add dataset_id to list of ids:
141         dataset_ids.push tf.dataset_id
142         # Reinitialize query_fields:
143         query_fields = Array.new
144         # Reinitialize query_table:
145         query_table = tf.table_name
146       end
147
148       # Add field name and transformed name to query:
149       query_fields.push tf.original_name + " AS " + tf.
             name
150
151       if tf == these_fields.last
```

```
152         queries.push "SELECT " + query_fields.join(",")
                + " FROM " + query_table
153         cache_tables.push "cache_table_" + params[:id]
               + "_" + query_table
154         if this_export.cache_type == 1
155           # Add where clause:
156           # query += " WHERE " + key_field + " > " +
                 max_val
157         end
158         query_fields = Array.new
159       end
160     end
161
162     this_dataset = nil
163     j = dataset_ids.length - 1
164
165     ### Refresh cache flag:
166     refresh_cache = false
167
168     ### Cache-only query flag:
169     cache_query = true
170
171     ### Determine if cache needs to be refreshed or not
           :
172     begin
173       if params[:cache_refresh] == "true" or
             this_export.cache_dirty or this_export.
             cache_expiration < Time.now
174         refresh_cache = true
175       end
176     rescue
177     end
178
179     if this_export.cache_type.to_i > 0 or joins.length
           > 0
```

```
180       cache_query = true
181     else
182       cache_query = false
183     end
184
185     if refresh_cache or not cache_query
186
187       @debug_msg += "<br />Reading Records"
188
189       ####
190       #### Read records:
191       ####
192       for i in (0..j)
193         if this_dataset == nil or this_dataset.id !=
             dataset_ids[i]
194           DbAccess.clear_active_connections!
195           DbAccess.clear_reloadable_connections!
196           DbAccess.remove_connection
197
198           this_dataset = Dataset.find(dataset_ids[i])
199           DbAccess.establish_connection(
200               :adapter  => this_dataset.connection_type
                    ,
201               :host     => this_dataset.connection_host
                    ,
202               :database => this_dataset.connection_name
                    ,
203               :username => this_dataset.connection_user
                    ,
204               :password => this_dataset.connection_pass
205           )
206         end
207
208         query_table_name = queries[i].sub( /^.+FROM / ,
             "")
```

```
209         cache_table_name = "cache_table_" + params[:id]
                + "_" + query_table_name
210
211         DbAccess.table_name = query_table_name
212
213         data_results = nil
214
215         data_results = DbAccess.find_by_sql( queries[i]
                );
216
217         if data_results != nil
218
219           if true or refresh_cache
220
221             ### Save cache state:
222             this_export.cache_dirty = false
223             this_export.cache_expiration = Time.now +
                   7.days
224             this_export.save
225
226             ###
227             ### Create cache tables:
228             ###
229             begin
230               ActiveRecord::Migration.drop_table
                     cache_table_name
231             rescue
232               ### Probably means the table does not
                     exist.. we're OK.
233             end
234             begin
235               ActiveRecord::Migration.create_table
                     cache_table_name do |t|
236                 data_results[0].attributes.each do |col
                        |
```

```
237                 this_field = MetadataField.find( :
                       first, :conditions => ["
                       unique_name=?",this_dataset.id.
                       to_s+"-"+query_table_name+"-"+col
                       [0]] );
238                 t.column col[0], this_field.
                       field_type
239               end
240             end
241           rescue
242           end
243
244           ###
245           ### Cache part:
246           ###
247           # cache_tables.push cache_table_name
248
249           Result.set_table_name(cache_table_name)
250           Result.reset_column_information
251
252           data_results.each  do |dr|
253             attr_hash = Hash.new
254             show_record = true
255
256             dr.attributes.each do |da|
257               ### Here is where we can apply
                     transforms:
258               transform_id = 0
259               if data_transforms[query_table_name+"."
                     +da[0]] != nil
260                 transform_id = data_transforms[
                       query_table_name+"."+da[0]].to_i
261               end
262               if transform_id.to_i > 0
```

```
263                 attr_hash[da[0]] = apply_transform(
                       da[1], transform_id )
264                 if attr_hash[da[0]] == false
265                   show_record = false
266                   break
267                 end
268               else
269                 attr_hash[da[0]] = da[1]
270               end
271             end
272
273             if show_record
274               if cache_query
275                 new_cache_record = Result.new(
                       attr_hash)
276                 new_cache_record.save
277               else
278                 new_data_results.push attr_hash
279               end
280             end
281           end
282         end # if refresh_cache
283       end
284     end
285   end
286
287   if cache_query
288
289     joins.each do |j|
290       Result.set_table_name("cache_table_"+params[:id
            ]+"_"+j.table1_name)
291       Result.reset_column_information
292       query = "SELECT * FROM cache_table_"+params[:id
            ]+"_"+j.table1_name+" t1 INNER JOIN
            cache_table_" + params[:id]+"_"+j.
```

```
            table2_name+ " t2 ON t1."+j.field1_name+"=t2
            ."+j.field1_name
293         new_data_results += Result.find_by_sql( query )
            ;
294         @debug_msg += "<br />Query: " + query
295         cache_tables.delete("cache_table_"+params[:id]+
            "_"+j.table1_name);
296         cache_tables.delete("cache_table_"+params[:id]+
            "_"+j.table2_name);
297       end
298
299       cache_tables.each do |c_table|
300         Result.set_table_name(c_table)
301         Result.reset_column_information
302         query = "SELECT * FROM " + c_table
303         @debug_msg += "<br />Query: " + query
304         new_data_results += Result.find_by_sql( query )
            ;
305       end
306
307     end
308
309     unless params[:time_test] == "true"
310       render :xml => new_data_results
311     end
312
313     @export_name = this_export.name
314     @cache_used = cache_query
315     @cache_updated = refresh_cache
316     @cache_expiration = this_export.cache_expiration
317     @end_time = Time.now
318     @num_records = new_data_results.length
319     @num_joins = joins.length
320
321   end
```

```
322
323   #
324   # Add field(s) to an export
325   #
326   def select_exported_field
327
328     @show_dataset_select = true
329     @selected_dataset = ""
330     if request.post?
331       @show_dataset_select = false
332       @selected_dataset = Dataset.find(params[:
            selected_field][:dataset_id].to_i).name
333     else
334       @datasets = Dataset.find(:all)
335       @dataset_options = {}
336       @datasets.each do |ds|
337         @dataset_options[ds.name] = ds.id
338       end
339     end
340
341     @all_done = false
342     @field_list = Array.new
343
344     if params[:export_id]
345         @export_id = params[:export_id]
346     else
347         @export_id = params[:selected_field][:export_id
            ]
348     end
349
350     #### See if dataset id is submitted:
351     if request.post?
352       if params[:selected_field][:dataset_id]
353
354         @selected_fields = Array.new
```

```
355           existing_fields = ExportedField.
                 find_all_by_export_id(params[:selected_field
                 ][:export_id], :select => [ "field_id" ] )
356           existing_fields.each do |ef|
357             @selected_fields.push ef.field_id
358           end
359           @field_list = MetadataField.find(:all, :
                 conditions => [ "dataset_id=?", params[:
                 selected_field][:dataset_id].to_i ],
360                                                 :order => "
                                                    table_name"
                                                     )
361         end
362
363         @blorg = ""
364         if params[:selected_field_fields] != nil
365           @all_done = true
366           params[:selected_field_fields].each do |field|
367             @blorg += field  + "<br>"
368             @metafield = MetadataField.find(field)
369             @exported_field = ExportedField.new( { :
                    field_id => field,
370                   :export_id  => @export_id,
371                   :field_name => @metafield.name,
372                   :field_type => @metafield.field_type,
373                   :transform  => "",
374                   } )
375             if @exported_field != nil and @exported_field
                    .save
376               @blorg += " - GOOD!<br />"
377             else
378               @all_done = false
379               #### Set error
380               break
381             end
```

```
382         end
383       end
384
385       #### If all is done, redirect to
              add_exported_field:
386       if @all_done
387         redirect_to( { :action => "edit_export", :id =>
              @export_id } )
388       end
389     end
390
391   end
392
393   #
394   # Create a new exported field
395   #
396   def add_exported_field
397     if request.post?
398       @exported_field = ExportedField.new(params[:
              exported_field])
399       if @exported_field.save
400         export = Export.find(@exported_field.export_id)
401         export.cache_dirty = true
402         export.save
403         redirect_to( { :action => "edit_export", :id =>
              params[:exported_field][:export_id] } )
404       end
405     end
406   end
407
408   #
409   # Delete an exported field:
410   #
411   def delete_exported_field
412     if request.post?
```

```
413       exported_field = ExportedField.find(params[:id])
414
415       export = Export.find(exported_field.export_id)
416       export.cache_dirty = true
417       export.save
418
419       exported_field.destroy
420     end
421     redirect_to(:action => :edit_export, :id => params
            [:export_id] )
422   end
423
424   #
425   # Edit an exported field:
426   #
427   def edit_exported_field
428     @exported_field = ExportedField.find(params[:id])
429
430     transforms = Transform.find(:all)
431
432     @transform_list = Hash.new
433     transforms.each do |trans|
434       @transform_list[trans.name] = trans.id.to_s
435     end
436
437     if request.post?
438
439       @exported_field = ExportedField.update(params[:id
            ], params[:exported_field])
440       if @exported_field.save
441         export = Export.find(@exported_field.export_id)
442         export.cache_dirty = true
443         export.save
444         flash.now[:notice] = "Export <b>#{
              @exported_field.field_name}</b> updated"
```

```
445       else
446         flash.now[:error] = "Error updating exported
              field"
447       end
448     end
449
450   end
451
452   #
453   # Show all exported fields:
454   #
455   def list_exported_fields
456     @all_exported_fields = ExportedField.find(:all)
457   end
458
459   #
460   # Show all transforms:
461   #
462   def list_transforms
463     @all_transforms = Transform.find(:all)
464   end
465
466   #
467   # Create a new transform:
468   #
469   def add_transform
470     @transform = Transform.new(params[:transform])
471     if request.post? and @transform.save
472       flash.now[:notice] = "Transform <b>#{@transform.
            name}</b> created"
473       redirect_to( { :action => "edit_transform", :id
            => @transform.id } )
474     end
475   end
476
```

```
  #
  # Remove a transform:
  #
  def delete_transform
    if request.post?
      transform = Transform.find(params[:id])

      export = Export.find(transform.export_id)
      export.cache_dirty = true
      export.save

      transform.destroy
    end
    redirect_to(:action => :list_transforms)
  end

  #
  # Update a transform:
  #
  def edit_transform
    if request.post?
      @transform = Transform.update(params[:id], params
         [:transform])
      if @transform.save
        flash.now[:notice] = "Transform <b>#{@transform
           .name}</b> updated"
      else
        flash.now[:error] = "Error updating export"
      end
    else
      @transform = Transform.find(params[:id])
    end
  end

  #
```

```
510   # Add an export join:
511   #
512   def add_export_join
513
514     fields = ExportedField.find( :all, :conditions=>["
          export_id=?",params[:export_id]] )
515     tables = MetadataField.find( :all, :conditions=>["
          id IN (?)",fields.map{|f| f.field_id} ], :select
          =>"DISTINCT table_name" )
516     @table_names = tables.map{|t| t.table_name}
517     @table_fields = MetadataField.find( :all, :
          conditions=>["id IN (?)",fields.map{|f| f.
          field_id} ], :order=>"table_name,name" ).map{|tf
          | [ tf.name, "("+tf.table_name+")."+tf.name ] }
518     @table_fields_hash = Hash.new
519     @table_fields.each do |tf|
520         @table_fields_hash[tf[1]] = tf[0]
521     end
522
523     @export_join = ExportJoin.new(params[:export_join])
524     if request.post? and @export_join.save
525       export = Export.find(@export_join.export_id)
526       export.cache_dirty = true
527       export.save
528       flash.now[:notice] = "Join created"
529       redirect_to( { :action => "edit_export", :id =>
            @export_join.export_id } )
530     end
531   end
532
533   #
534   # Delete an export join:
535   #
536   def delete_export_join
537     if request.post?
```

```
538       join = ExportJoin.find(params[:id])
539
540       export = Export.find(join.export_id)
541       export.cache_dirty = true
542       export.save
543
544       join.destroy
545     end
546     redirect_to(:action => :edit_export, :id => params
            [:export_id] )
547   end
548
549   def test_transform
550     @transform = Transform.find(params[:id])
551     @output = ""
552     @test_code_block = @transform.code_block
553
554     if request.post?
555       code = "input = \"" + params[:test_value] + "\""
556
557       ### Determine input type:
558       if @transform.input_type.downcase == "integer"
559         code += ".to_i"
560       elsif @transform.input_type.downcase == "float"
561         code += ".to_f"
562       end
563
564       code += "\n" + params[:test_code_block]
565       @test_code_block = params[:test_code_block]
566
567       ### Determine return type:
568       if @transform.output_type.downcase == "integer"
569         code += "\ninput.to_i"
570       elsif @transform.output_type.downcase == "float"
571         code += "\ninput.to_f"
```

```
572       elsif @transform.output_type.downcase == "boolean
              "
573         # code += "\ninput.to_b"
574       else
575         code += "\ninput.to_s"
576       end
577
578       @output = eval code
579       # @output = code
580     end
581   end
582
583   ###
584   ### Transformation tool:
585   ###
586   def apply_transform( val, transform_id )
587     ### Lookup transform by ID:
588     transform = Transform.find( transform_id )
589     code = "input = \"" + val.to_s + "\""
590
591     ### Determine input type:
592     if transform.input_type.downcase == "integer"
593       code += ".to_i"
594     elsif transform.input_type.downcase == "float"
595       code += ".to_f"
596     end
597
598     code += "\n" + transform.code_block
599
600     ### Determine return type:
601     if transform.output_type.downcase == "integer"
602       code += "\ninput.to_i"
603     elsif transform.output_type.downcase == "float"
604       code += "\ninput.to_f"
605     elsif transform.output_type.downcase == "boolean"
```

```
606       # code += "\ninput.to_b"
607     else
608       code += "\ninput.to_s"
609     end
610 
611     if transform.output_type.downcase == "boolean"
612       if eval code
613         return val
614       else
615         return false
616       end
617     else
618       return eval code
619     end
620   end
621 
622 end
623 ### End webservice_controller ###
```

BIBLIOGRAPHY

[1] *PostgreSQL 8.2.1 Documentation*, 2006.

[2] Altinal, M., Bronhovd, C., Krishnamurthy, S., Mohan, C., Pirahesh, H., and Reinwald, B. Cache tables: Paving the way for an adaptive database cache. In *Proceedings of the 29th VLDB Conference* (2003).

[3] Banzhaf, H. S. Establishing a bureau of environmental statistics: more data collection and analysis would greatly enhance our ability to set policy and measure its effectiveness. *Issues in Science and Technology 20.2* (2004), 25–6.

[4] Bohr, J., and Zybtowski, J. Michigan fish contaminant monitoring program (annual report).

[5] Bontempo, C., and Zagelow, G. The ibm data warehouse architecture. *Communications of the ACM 41*, 9 (September 1998), 38–48.

[6] Cooper, P. *Beginning Ruby: From Novice to Professional*. Apress, 2007.

[7] Curbera, F., Khalaf, R., Mukhi, N., Tai, S., and Weerawarana, S. Service-oriented computing: The next step in web services. *Communications of the ACM 46*, 10 (October 2003), 29–34.

[8] Daconta, M. C., Obrst, L. J., and Smith, K. T. *The Semantic Web*. Wiley, 2003.

[9] El Maghraoui, K., Desell, T. J., Szymanski, B. K., and Varela, C. A. The internet operating system: Middleware for adaptive distributed computing. *The International Journal of High Performance Computing Applications 20*, 4 (Winter 2006), 467–480.

[10] Ikoma, E., Taniguchi, K., Koike, T., and Kitsuregawa, M. Development of a data mining application for huge scale earth environmental data archives. *Int. J. Computational Science and Engineering 2* (2006), 262–270.

[11] Issarny, V., Caporuscio, M., and Georgantas, N. A perspective on the future of middleware-based software engineering. In *International Conference on Software Engineering* (2007), IEEE Computer Society.

[12] Jarke, M., Lenzerini, M., Vassiliou, Y., and Vassiliadis, P. *Fundamentals of Data Warehouses*. Springer, 2000.

[13] Jukic, N. Modeling strategies and alternatives for data warehousing projects. *Communications of the ACM 49*, 4 (2006), 83–89.

[14] Kantardzic, M. *Data Mining: Concepts, Models, Methods, and Algorithms*. Wiley-Interscience, 2003.

[15] Liu, H., and Motoda, H., Eds. *Instance Selection and Construction for Data Mining*. Kluwer Academic Publishers, 2001.

[16] Milo, T., Abiteboul, S., Amann, B., Benjelloun, O., and Ngoc, F. D. Exchanging intensional xml data. *ACM Transactions on Database Systems 30*, 1 (March 2005), 1–40.

[17] Sen, A., and Sinha, A. P. A comparison of data warehousing methodologies. *Communications of the ACM 48* (2005), 79 – 84.

[18] Singh, H. *Data Warehousing: Concepts, Technologies, Implementations, and Management*. Prentice-Hall, 1998.

[19] Thomas, D., Hansson, D., Breedt, L., Clark, M., Gehtland, J. D. D. A. J., and Schwarz, A. *Agile Web Development With Rails*, 2nd ed. Pragmatic Bookshelf, 2006.

[20] Wang, J., Ed. *Encyclopedia of Data Warehousing and Mining*, vol. 1. Idea Group Reference, 2006.

[21] Wang, J., Ed. *Encyclopedia of Data Warehousing and Mining*, vol. 2. Idea Group Reference, 2006.

[22] WN Venables, D. S. *An Introduction to R*, 2008.

[23] Ye, N., Ed. *The Handbook of Data Mining*. Lawrence Erlbaum Associates, Inc, 2003.

[24] Zhang, L.-J. *Modern Technologies in Web Services Research*. IGI Publishing, 2007.