Metadata Management for Federated Databases

Carlos Ordonez
University of Houston
Houston, TX, USA

Zhibo Chen
University of Houston
Houston, TX, USA

Javier García-García
UNAM University
Mexico City, Mexico

ABSTRACT

A federated database consists of several loosely integrated databases, where each database may contain hundreds of tables and thousands of columns, interrelated by complex foreign key relationships. In general, there exists a lot of semistructured data elements outside the database represented by documents (files), created and updated by multiple users and programs. Documents have references to multiple databases and subsets of their tables and columns. Manually tracking which specific tables and columns are referred to by a document, accessed by a specific program or user is a daunting task. With such a goal in mind, we present a system that builds metadata models for a federated database using a relational database as a central object type and metadata repository. Metadata includes table and columns coming from logical data models corresponding to each database, as well as documents representing external semistructured data sources. SQL statements assemble metadata and data types to create objects and relationships as relational tables for easy querying. We discuss potential applications in federated scientific databases.

1. INTRODUCTION

Metadata management in relational databases is important to understand how tables and columns are exploited by users. In this work, we focus on metadata management for a federated database environment. In general, a significant portion of metadata (knowledge about the database) is stored outside the database, and has generally an unstructured [3] or semistructured form [5]. We will use the term semistructured to refer to both forms. Such metadata elements include table and column names from the logical data model (LDM), application programs, specification documents, diagrams, spreadsheet files, images, web pages and so on. In metadata management the overall goal is to capture information about relationships among components in each database, as well as linking them to end users and application programs. In our proposal, we combine object-oriented technology and relational databases to integrate and manage metadata. Traditionally, it has been a common practice for scientists to work in isolation on their experiments, creating private data repositories. Nowadays, with the growing usage of the Internet and faster computers, more scientists are interested in taking advantage of technology to make their data sets and documents widely available. Scientists exchange and share data sets and documents with other scientists to compare results or enhance their experiments. This scenario creates many challenges in managing interrelated structured and semistructured data. Many proposals have tackled the problem from the information retrieval perspective. In our work, we propose to solve this problem with relational database technology.

We believe using a relational database as a central metadata repository for structured and semistructured information is the right approach. The database serves three goals: (1) it acts a metadata repository that stores both structured and semistructured information in tabular form; (2) it centralizes metadata management for multiple databases; (3) it stores object type definitions and valid (allowed) relationships among them. Managing a large metadata database is difficult when there exist multiple databases and complex relationships among thousands of objects of different types. Some interesting and challenging aspects include the following. Metadata from different source databases needs to be loaded, transformed and integrated. Semistructured data must be cleaned and transformed into a tabular form, appropriate to be imported into a relational database. For documents this implies extracting keywords. Table structure for similar information elements may be different across databases. Database content may be incomplete and inconsistent, especially for semistructured data. Data types for objects and relationships must be stored in tabular form, whereas a tree data structure would be a better fit. The model must allow the definition of complex relationships among objects including tables, columns, users, programs and documents. In this article we present a proposal to solve these problems.

This work contains three main contributions. First, we motivate and explain how to integrate metadata from federated databases and related semistructured data sources. We also introduce a simple approach to transform unstructured data into a tabular form to be imported into the database. Second, we explain how to use a relational database to store object data types, valid relationships among objects and metadata from semistructured and structured data sources. Third, since metadata and data type information are stored in relational tables then SQL becomes the alternative to define, build and link objects. We explain how to write SQL queries for such purpose.

The article is organized as follows. Section 2 introduces basic concepts on semistructured and structured data and on metadata and object oriented models. Section 3 explains

http://doi.acm.org/10.1145/1317353.1317361

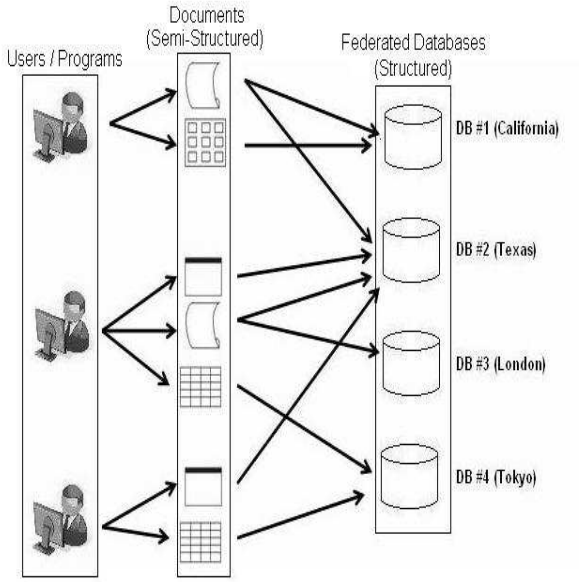

Figure 1: Federated database environment.

how to use a relational database as a metadata repository and how to assemble objects and relationships to build metadata models linking semistructured and structured data. Section 4 discusses closely related research. Section 5 contains the conclusions and directions for future work.

2. PRELIMINARIES

Structured data sources are represented by all databases in the federated database. Information elements in databases are basically table and column names, to be manipulated as metadata. Semistructured data sources are represented by documents. Documents are files of several kinds including digital documents, word processing files, spreadsheets, text files, web pages and email communications. To make exposition simpler we do not make a distinction between the logical data model (LDM), represented by an ER [4] diagram and the physical data model (PDM), represented by actual tables and columns in a DBMS. The diagram in Figure 1 shows a federated database environment with both semistructured and structured data sources.

We use an object-oriented (OO) model to represent metadata elements [2] and a relational database [4] as a metadata repository. The object-oriented model imposes a correct structure definition of objects and a specification of valid relationships among them. We now introduce basic object oriented terminology. The basic modeling elements are objects, containers and relationships and each of these elements has a data type. Containers are used to place objects into meaningful groups. We consider two basic data types to build and link objects: an object type and a relationship type. An object type defines an object structure (properties), and containment relationships with other objects (objects part of other objects). A relationship type defines a pair of object types that can be linked (related) to each other. We focus

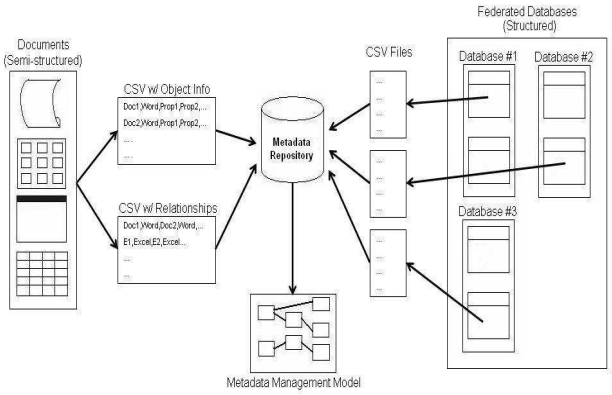

Figure 2: Data sources flow.

on binary relationships; higher cardinality relationships are expressed as a collection of binary relationships. Each object type, each existing object and each relationship constitute a link. This implies a hierarchical (tree structured) object organization. Object data types are organized into type libraries. All databases, tables, columns, users, programs and documents are manipulated as objects.

3. RELATIONAL METADATA MODELS

This section presents our main contributions. We start by presenting research issues. We then discuss our system architecture. We explain how to import unstructured data sources. We explain how to assemble object and relationships. We finally discuss application with scientific databases.

3.1 Research issues

We have identified four main challenges:

1. Automating the transformation of unstructured and semistructured data into a tabular form. This includes preprocessing documents to extract keywords.

2. Dealing with inconsistency and incompleteness in metadata. This is more difficult in semistructured data sources.

3. Managing data types, especially for relationships among objects.

4. Creating links among objects through relationships. The fundamental issue is managing relationships among structured and semistructured objects.

3.2 System Architecture

Structured and semistructured data sources feed a central metadata repository. All logical data models from the databases represent structured sources. Documents represent semistructured sources. Users and programs update documents. For all purposes object information is imported into the database using text files, as explained below. The basic system architecture to build a metadata management model is shown in Figure 2.

Semistructured data sources include ER diagrams, word processor documents, spreadsheets, application names, documentation about programs, and so on. Such information must be converted to tabular form using CSV text files. The database is understood as a collection of interrelated tables containing information in the form of objects, relationships and meta-data defining objects structure. The basic mechanism to store and retrieve information in the relational database is the SQL language [4].

3.3 Importing Data Sources with CSV Files

This section explains how to import data from diverse sources into a relational database using CSV text files. Data elements come from semistructured sources [8, 5] including Entity-Relationship model diagrams, word processor documents, spreadshseet files, e-mail messages and so on. In general most tools provide mechanisms to convert any type of file to a CSV file or something that is quite similar. If not available it is easy to export information to a spreadsheet or text editor and then export the information as CSV files.

CSV files Overview

We use Comma Separated Values (CSV) text files as the basic mechanism to import data into the relational database. CSV files have several remarkable advantages. CSV files are a de-facto standard for many computer tools. CSV files are self-describing because both data element names and data instances are stored together, making them a simpler alternative to XML. CSV files can be easily imported into relational tables. They do not require complex data formatting with specific positions and fixed lengths. They are self-contained (no extra files needed). They do not waste space as formatted files do. A person can easily manipulate them in any spreadsheet program, editor or word processor. They are portable between UNIX and Windows platforms. They can easily manage missing information because they do not need a special symbol to mark missing values; two consecutive value separators indicate a missing value. Most non-database tools can generate them. Many times they are the only way to export/import data. A CSV file can be understood as a table that can be imported into a relational database. Each column contains homogeneous information. Each row contains elements of different data types. Any type of information can be manipulated with CSV files. The only constraint is that each element must be a number or a string. This makes them adequate to manipulate documents, data models, spreadsheets, e-mail messages and so on. Nevertheless, CSV files are not the ultimate answer to data exchange. Some important issues include the following. In general they are isolated (i.e. not related/linked to other CSV files). There may be different number of values per line, although in many cases it is equal. Each element in the list (in one line) does not necessarily have the same type as the corresponding elements in other lines. A string can contain a comma inside. That is, it may contain the value separator. A string may or may not be delimited by double quotes (”). The type of each value is determined on an individual value basis. Compared to XML [13] CSV files are simpler, but of course, they provide a more primitive structure (tabular versus hierarchical).

CSV files are well understood, but give a brief technical description. The CSV file is a text file without any special control characters but end of line and end of file. Each line has a list of values separated by commas and ends with end of line. Comma and end of line are the only permitted value separators. Each value can be treated as a number or a string. Strings are sequences of symbols that may or may not be delimited by quotes. We assume CSV files contain a header line with column names.

Defining a table and inserting rows

Our system takes CSV files as input and automatically creates the definition for a table to store imported data. This creation takes care of determining the best data type for each column, computing lengths for strings of text, and automatically creating column names when not available. But in general each csv file is assumed to have a header with column names. The column data type is determined by a voting algorithm that scans each line in the CSV file and consider three basic data types: integers, floating point numbers and strings. Missing values are ignored for determining data types. To avoid missing rows we follow a pessimistic approach: (1) a column where most values are numbers but has strings it is taken as string. (2) if a column with integers has at least one floating point] number the data type is floating point. The maximum counter determines the column type. If the column is a string the size is determined by the longest string found in that column. The maximum number of values contained in any line determines the table number of columns. Since row uniqueness cannot be assumed a surrogate primary key (an id) is automatically created. In some cases, the first column is taken as the default primary index to access the table.

Once a table structure has been defined we need to convert CSV file text lines into SQL insert statements. The basic idea is that each element in line becomes an argument to the insert statement and the comma remains a separator. However, several details are important. Missing values are automatically taken care of; there is no need to explicitly use null values. String delimiters in the input are double quotes ”, whereas SQL string delimiters are single quotes. All strings must have delimiters to be used in SQL. There may be a different number of elements per line and the program must make sure each insert statement has exactly the same number and that such number matches the table definition. In the exceptional case when there are string delimiters inside a string they must be preceded by an escape sequence. In this case the quote character ’ is repeated. Since we expect to have no more than thousands of objects for knowledge management SQL inserts provide reasonable performance.

3.4 Object Types and Relationship Types

The basic elements are objects, containers and relationships and each of these elements has a data type. We use the following rules to build and link objects. A pair {object type,object name} must be unique across the model. Therefore, it represents a primary key. An object can be linked to another object if and only if there exists a predefined relationship type that links both object types; this leads to defining a table with five columns, where two object types act as foreign keys. Objects are grouped into containers, providing a hierarchical organization. Containers can contain any number of containers and they can be recursively nested. Objects can contain objects, but they cannot have containers. Objects can be linked by relationships across container boundaries, but containers cannot be

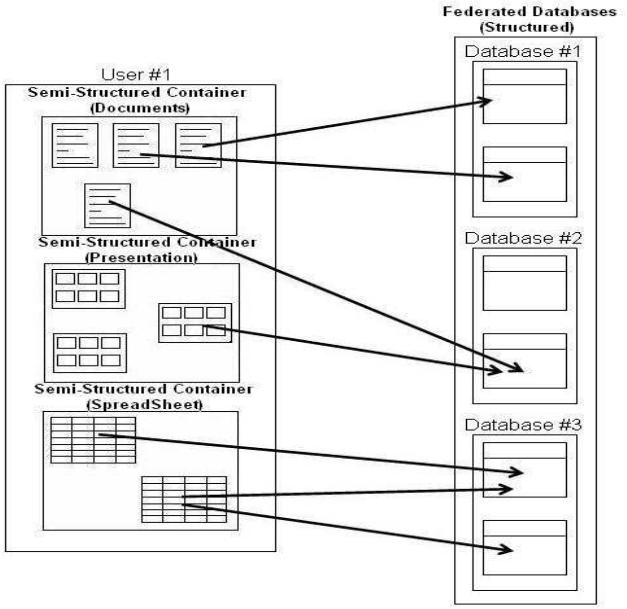

Figure 3: Macro-relationships between semistructured and structured data sources.

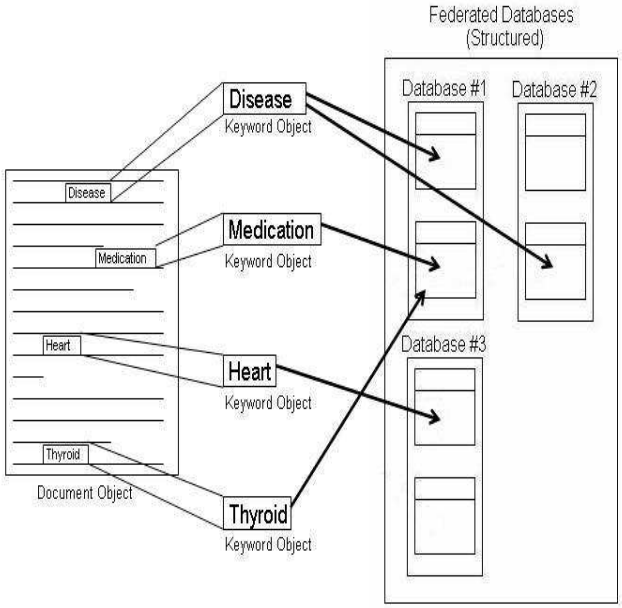

Figure 4: Micro-relationships between keywords and tables/columns.

linked to each other in any manner. The diagram in Figure 3 shows relationships between objects at a coarse granularity level. The diagram in Figure 4 shows relationships between objects at a fine granular level, linking atomic elements between semistructured and structured data sources.

We use a relational DBMS as a repository for object types, sets of instantiated objects and valid relationships among them. Each object has the following minimal set of properties: type, name and description. The object's name and type are used together as a primary key to identify objects. The relationship primary key is given by the two objects it relates and a data type of its own.

As mentioned above, an object is identified by its type and its name. The same object name can have two or more types. So a single name may effectively represent two or more distinct objects. Special care must be taken in interpreting object names. In general the object type must also be considered together with the object name. A relationship between two objects is identified by the primary keys corresponding to the objects and the relationship type. Containers are object place holders and cannot be related to each other like objects. Our proposal assumes containers are defined by the end-user to appropriately group objects.

There are three tables to manage object type information (metatypes). The first table stores detailed object type information. The second table stores information for type manipulation including a type library name, a description and users maintaining it. The third table links an object type with a type library and specifies a level for object nesting. The three tables are referentially related by the library and type names. This is a normalized schema that provides flexibility to change object types from one library to another as needed and simplifies SQL code generation. A summary of these tables is shown in Table 1.

We use two generic tables (meta-tables) to store object information. In general, information coming from different sources is stored in denormalized tables that are structurally similar to either of these tables. For each object there is a set of properties stored as columns that is stored in a table for objects having the same type. In general, such table contains information only for one object type, but in some cases this table contains object names and properties for two or more types of objects storing some column values as null. A "link" table stores multiples relationships in many pairs of objects. The database has a collection of object instance tables and a collection of object relationship tables. In general, metadata objects coming from external sources come unrelated and they must be manually linked by defining a relationship. Relationships are specified inside the database by treating two object types as foreign keys, linking two object types with a relationship type and enforcing referential integrity. The diagram in Figure 5 shows how object types and relationships are combined to instantiate a new object.

3.5 SQL Metastatements

We use three template SQL statements (metastatements) to create (instantiate) objects and link them with given relationships. The first SQL statement creates an object and its properties. It assembles object name, type and properties together. This query requires duplicate removal, one natural join and deleting rows with null values. This statement can also be used when there is a need to modify a set of nested objects of the same type or a relationship between two sets of objects of respectively the same type. The second SQL statement creates nested objects. That is, objects contained inside other objects. Both the nesting level and

Table name	Primary key	Columns
typeLibrary	library	libraryDescription,libraryPerson,dateModified
typeInfo	typeName,library	creationDate,modifyDate
typeControl	typeName	ObjectOrRelationship,typeDescription, defaultLibrary,sourceFile,objectLevel

Table 1: Tables to manage object types.

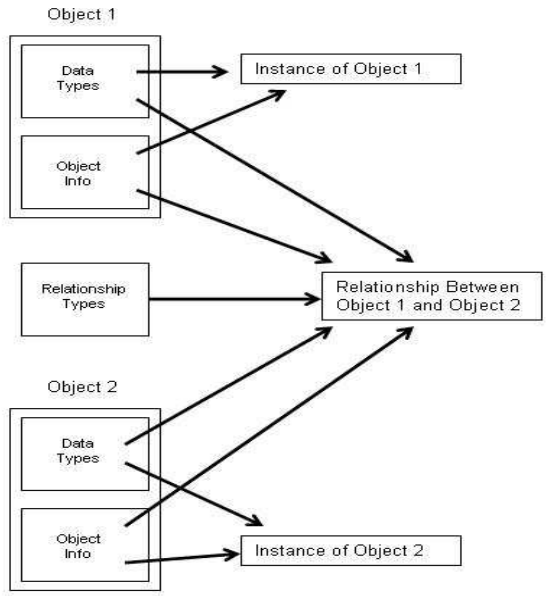

Figure 5: Creating relationships.

object containment need to be specified. If there are n levels of object nesting there will be $n-1$ SQL statements specifying nesting between consecutive levels. The third SQL statement creates a relationship between two objects and is the most complex. The relationship goes in one direction from a source object to a target object. Three data types need to be specified: the type for the source object, the type for the target object and the type for the relationship itself. Also, we need to give the name of the source and target objects giving a total of five attributes to identify a relationship. This statement requires defining a table where the relationship between both objects is implicitly assumed when their names appear on the same row. These three SQL statements cover every potential case encountered when creating metadata models.

The first type of SQL statement is the simplest. It is used to export an object and its properties. This just requires gathering the object name, type and properties and joining them. The information of *objectInfo* is assembled together with type information. This statement requires duplicate row removal, one natural join and deleting rows with null values. The generic SQL is as follows:

```
SELECT DISTINCT
  objectType,objectName,property1,...,propertyN
FROM   objectInfo,typeControl,typeInfo
WHERE  typeDescription='description'
  AND typeControl.typeAbbreviation
     =typeInfo.abbreviation
  AND typeControl.defaultLibrary
     =typeInfo.library
  AND objectName IS NOT NULL;
```

The second type of SQL export statement involves nested objects. That is, objects contained inside other objects. In this case the SQL code is more complex because the nesting level and object containment need to be specified. Because of modeling tool constraints rows need to appear in a certain order to specify object nesting. This statement requires duplicate row removal, joins, a set union, derived tables for parent/child objects, child object statement appearing before than parent object statement and result ordering. The objects are grouped by parent object. Notice a parent object is parent of itself as well in order to have a uniform table scheme. If there are n levels of object nesting there will be $n-1$ SQL statements specifying nesting between consecutive levels. The lowest level corresponds to the root and levels must appear in order of depth in the tree. The generic SQL is as follows:

```
SELECT
 level,type,name
FROM(
 SELECT DISTINCT
    objectLevel,parentObjType
   ,parentObjName,childObjName
 FROM   relatedObjects,typeControl,typeInfo
 WHERE  typeDescription='childObject'
   AND  typeControl.typeAbbreviation
       =typeInfo.abbreviation
   AND  typeControl.defaultLibrary
       =typeInfo.library
   AND childObjName  IS NOT NULL
   AND parentObjName IS NOT NULL
 UNION
 SELECT DISTINCT
    objectLevel,parentObjType
   ,parentObjName,parentObjName
 FROM   relatedObjects,typeControl,typeInfo
 WHERE  typeDescription='parentObject'
   AND  typeControl.typeAbbreviation
       =typeInfo.abbreviation
   AND  typeControl.defaultLibrary
       =typeInfo.library
   AND  parentObjName  IS NOT NULL
) tempTable(level,type,parentname,objname);
```

The third statement type is the most complex. It involves creating a relationship between two objects. The relationship goes in one direction from a source object to a target object. Three data types need to be specified: the type for

the source object, the type for the target object and the type for the relationship itself. Also, we need to give the name of the source and target objects giving a total of five attributes to identify a relationship. This statement requires a *relatedObjects* table where the relationship between both objects is implicitly assumed when their names appear on the same row. This statement requires derived tables, duplicate row removal in derived tables, a self-join, natural joins and careful string manipulation. The most important differences with respect to the previous SQL statements are the self-join which is required to manipulate relationships and trimming spaces to make sure object names exactly match. The generic SQL is as follows:

```
SELECT DISTINCT
  source.ObjectType,source.ObjectName
 ,relationshipType
 ,target.ObjectType,target.ObjectName
FROM
 (SELECT DISTINCT dataType1,trim(object1)
   FROM   relatedObjects,typeInfo,typeControl
  WHERE  typeDescription='object1'
    AND  typeControl.typeName
        =typeInfo.typeInfo.typeName
    AND  typeControl.defaultLibrary
        =typeInfo.library
 ) source(ObjectType,ObjectName)
,(SELECT DISTINCT dataType2,trim(object2)
  FROM  relatedObjects,typeInfo,typeControl
  WHERE  typeDescription='object2'
    AND typeControl.typeName
       =typeInfo.typeName
    AND typeControl.defaultLibrary
       =typeInfo.library
 ) target(ObjectType,ObjectName)
,relatedObjects,typeControl,typeInfo
WHERE
  typeDescription='relationship object1-object2'
  AND relatedObjects.object1
     =source.ObjectName
  AND relatedObjects.object2
     =target.ObjectName
  AND typeControl.typeAbbreviation
     =typeInfo.abbreviation
  AND typeControl.defaultLibrary
     =typeInfo.library;
```

3.6 Enforcing Strong Type Checking

Relationships are enforced by the relational database using predefined types linking two object types. Modeling column names is difficult because they can be treated as independent objects that can be related to any table or as nested objects inside tables. In the first case the column name appears once and in the second case it may be replicated in different tables. Even though there is redundancy, we support nesting column names inside table names because that is most similar to the relational database representation. Therefore, to guarantee a unique object name the table name is concatenated with the column name. When creating relationships all source objects and all target objects must have the same data type, respectively; but source and target objects can have different types. When creating nested objects it is required to have separate SQL statements to specify nesting and to specify object properties because in general nested objects are of different types. The nesting statement specifies the level, the object type and the object name. The second statement complements the first statement by defining the remaining object properties. Strong type checking is essential to assure model correctness. Furthermore, correct relationship creation and object nesting are enforced through the use of type checking.

3.7 Defining Relationships

Tables in the same database are linked by foreign key relationships. We define a relationship between two tables in different databases if they have columns with similar content. We define a relationship between two columns in different tables, in different databases, if they have similar content. Application programs and users have relationships with multiple tables and columns. Such information comes from semistructured data elements (documents) like word processing files, spreadsheets, text files and so on. In general, this is the most challenging step because it is a semi-manual process and semistructured data is more inconsistent and incomplete than data from a relational database.

3.8 Application with Scientific Databases

First, consider an application in medical databases. Suppose doctors at several medical centers around the world decided to collaborate on their research by sharing their findings and data. Since these centers were independently operating before this collaboration, it is unlikely that they would have identical database systems with compatible storage methods. Instead, the combination of these centers would resemble a federated database system. In this setting, the proposed model would utilize the information stored within semi-structured sources (documents, spreadsheets, text files, etc) to create metadata that would be stored within a structured data central repository. Further processing would create links between semistructured metadata and specific tables/columns in the federated databases. The power of discovering such links lies in its usefulness in assisting researchers with locating potentially valuable data. It is infeasible that a researcher at Center A would know the details of information stored at other centers. Currently, that researcher would have to browse through a long list of all the data included within those unfamiliar databases. One can imagine how much time this must waste, especially in large databases. On the other hand, suppose that the proposed model was used and the centers had a repository with metadata linking documents to tables. These relationships would allow scientists to determine which databases were used by specific documents, or users. As such, researchers could now simply lookup fellow doctors who are studying similar topics and obtain knowledge regarding which databases and tables were visited by that doctor. This would not only significantly reduce the amount of time and effort wasted in locating potential data, but also increase the odds of finding valuable information. Tables 2 and 3 show example relationships for the medical database.

As a second example, let us consider a federated database to manage experiment data from researchers collaborating on a common project. We make the following assumptions. Researchers work at different locations and are connected by the Internet. Researchers have access to each other's databases and documents outside the database. The

DataType1	DataName1	Role	EmpID	DataType2	DataName2	Num of Rows
User	Anderson	Doctor	1267	Table	Heart	655
User	Johnson	Nurse	1578	Table	Thyroid	9873
User	Pham	Nurse	4523	Table	Medication	567200
User	Smith	Doctor	5326	Table	Heart	655
User	Smith	Doctor	5326	Table	Treatment	7640

Table 2: Inferred macro-relationships between users and tables.

DataType1	DataName1	Size	DataType2	DataName2	Type	RelationshipType
Keyword	Pressure	8	Column	Heart.BP	Float	Keyword-Column
Keyword	Diabetic	8	Column	Heart.Diab	Integer	Keyword-Column
Keyword	Dosage	6	Column	Medication.Dosage	Integer	Keyword-Column
Keyword	Cost	4	Column	Medication.Cost	Float	Keyword-Column
Keyword	Illness	7	Column	Patient.Illness	String	Keyword-Column

Table 3: Direct micro-relationships between keywords and columns.

databases have some overlapping content, but they are not integrated. There exists a large collection of documents around each database which have experiment descriptions with data extracted from the databases. Such data elements include measurements, experiment parameters and hypotheses. Now consider the following usages for a researcher accesing the centralized metadata repository. The user can find all documents referring to the same table representing some data set. The user can know equations used by other researchers involving a certain column in a table. It is possible to investigate who used tables with similar content across databases, enabling better interaction. The researcher can list programs that process data sets exported out of the database, and to understand which tasks are not commonly performed inside the database system. Linked documents can further help understanding how external programs process such data sets.

4. RELATED WORK

In our approach we exploit principles from object oriented databases [1], which are applied in relational database systems [12, 4]. Integrating data coming from a variety of sources is a well researched problem. There has been work on translating schema information to integrate heterogeneous information [7], integrating unstructured [3] and linking semistructured information [5] with databases.

Previous work on managing scientific data includes the following. A prototype system to manage large result files from numerical simulations is presented in [11]. The proposed interface was specifically designed for users with a scientific background and not familiar with SQL. By using an intuitive searching and browsing mechanism the user can store, search and retrieve large, distributed, files resulting from scientific simulations. The interface construction is automated and dynamic so that it requires little database/web development experience to install and use. This is achieved by allowing the user interface specification to be defined in an XML file used to initialize the system and by providing a program that can generate default XML specifications. An architecture for creating a collaborative centralized infrastructure for sharing scientific data is proposed in [6]. The architecture uses a search scheme that allows users to locate data sets by exploiting metadata information.

Our proposal is related to previous work on measuring and improving data quality in relational databases [9, 10]. In [10] referential integrity quality metrics are defined on a centralized database [10]. On the other hand, [9] studies the computation of consistent aggregations in the presence of referential integrity errors.

5. CONCLUSIONS

Our proposal is about building metadata management models for federated databases using a relational database as a central metadata repository. We use an object-oriented model to represent databases, tables, columns, documents, users and programs as objects linked by relationships. SQL is used to assemble objects and create relationships in object pairs, based on carefully defined data types. Once the metadata database has been built it is easy to query which documents, programs and users have references to specific tables and columns in multiple databases. Different databases can be indirectly related through relationships with semistructured data sources. We presented potential applications with scientific databases.

There are several issues for future work. We need to study how to organize and automate semistructured information more accurately. This requires preprocessing documents to extract information in tabular form and solving inconsistency issues. We need to study how to create data type hierarchies for semistructured information. We need to consider evolving logical data models where tables and columns are constantly added or removed.

Acknowledgments

The third author is with Universidad Nacional Autónoma de México (UNAM) and was sponsored by the UNAM IT project "Macroproyecto de Tecnologías para la Universidad de la Información y la Computación".

6. REFERENCES

[1] M. Atkison, F. Bancilhon, D.J. DeWitt, K. Dietrich, D. Maier, and S. Zdonik. The object oriented database system manifesto. In *DOOD*, pages 223–240, 1989.

[2] M. Blaha and W. Premerlani. *Object-Oriented Modeling and Design for Database Applications*. Prentice Hall, 1998.

[3] P. Buneman, S. Davidson, and M. Fernandez. Adding structure to unstructured data. In *ICDT Conference*, pages 336–350, 1997.

[4] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison/Wesley, Redwood City, California, 3rd edition, 2000.

[5] R. Goldman and J. Widom. Enabling query formulation and optimization in semistructured databases. In *VLDB Conference*, pages 436–445, 1997.

[6] A. R. Jaiswal, C. L. Giles, P. Mitra, and J. Z. Wang. An architecture for creating collaborative semantically capable scientific data sharing infrastructures. In *ACM WIDM Workshop*, pages 75–82, 2006.

[7] T. Milo and S. Zohar. Using schema matching to simplify heterogeneous data translation. In *VLDB Conference*, pages 122–133, 1998.

[8] S. Nestorov, S. Abiteboul, and R. Motwani. Extracting schema from semistructured data. In *ACM SIGMOD Conference*, pages 295–306, 1998.

[9] C. Ordonez and J. García-García. Consistent aggregations in databases with referential integrity errors. In *ACM IQIS Workshop*, pages 80–89, 2006.

[10] C. Ordonez and J. García-García. Referential integrity quality metrics. *Decision Support Systems Journal*, 44(2):495–508, 2008.

[11] M. Papiani, J.L. Wason, A.N. Dunlop, and D.A. Nicole. A distributed scientific data archive using the Web, XML and SQL/MED. *SIGMOD Rec.*, 28(3):56–62, 1999.

[12] W. Permelani, M. Blaha, and J. Rumbaugh. An object-oriented relational database. *Comm. ACM*, 33(11):99–109, 1990.

[13] J. Shanmugasundranam, H. Gang, K. Tufte, C. Zhang, D.J. DeWitt, and J.F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *VLDB Conference*, pages 302–314, 1999.