MANAGING XML DATA IN A RELATIONAL WAREHOUSE: ON QUERY TRANSLATION, WAREHOUSE MAINTENANCE, AND DATA STALENESS

By

RAJESH KANNA

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2001

To my parents, Kalia Perumal and Shanthi Perumal, and my brothers Ragu Ram and Shankar Ganesh.

ACKNOWLEDGMENTS

First and foremost, I would like to express my sincere gratitude to my advisor, Dr. Joachim Hammer for giving me an opportunity to work on this interesting topic and for providing me with excellent advice, great guidance and support throughout the course of this research work. His tireless patience and knowledge have been invaluable. Without his guidance, this thesis would not have been possible.

I wish to thank Dr. Abdelsalam Helal and Dr. Douglas Dankel for graciously agreeing to serve on my committee and for taking time out from their busy schedules to read and comment on my thesis.

My thanks go to Sharon Grant and Mathew Belcher for making the Research and Database Center a truly great place to work. It has been a great honor to be part of the IWiz Project Team. It was fun working with Charnyote Pluempitiwiriyawej, Anna Teterovskaya, Amit Shah and Ramasubramanian Ramani. I also thank Sivakumar Balakrishnan and Karthik Ramani who made me feel at home and were with me all the way.

Finally, I would like to thank my family back home for their unwavering moral support and endless love.

TABLE OF CONTENTS

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

MANAGING XML DATA IN A RELATIONAL WAREHOUSE : ON QUERY TRANSLATION, WAREHOUSE MAINTENANCE, AND DATA STALENESS

By

Rajesh Kanna

August 2001

Chairman: Joachim Hammer
Major Department: Computer and Information Science and Engineering

As more and more data is made available through the Web, mediation of information from heterogeneous sources becomes the focal point for future Web-based information systems. The Integration Wizard Project (IWiz) is a research prototype system under development at the University of Florida that provides integrated access to multiple, heterogeneous information sources using a combined mediation/data warehousing approach.

IWiz uses mediators and source wrappers to extract and integrate the information relevant to a user query from the available sources. Its internal data model is based on XML and DOM. Query results are also cached in a relational data warehouse to speed up similar queries in the future.

This thesis describes the underlying design and development of three critical components that are part of the data warehouse manager (WHM) of IWiz. Generally speaking, the WHM provides functionality for storing, querying, and maintaining the results of frequently submitted queries to IWiz using a relational database engine. Specifically, the components described in this research are as follows:

1. A decision module for determining whether a given user query can or should be answered using the contents in the data warehouse or whether it needs to be submitted to the sources;
2. An XMLQL-2-SQL query translator for converting user queries against the integrated, XML-based view of the underlying sources into equivalent queries against the relational warehouse schema; and
3. A maintenance query generator for monitoring the hit-rate of the data warehouse and for generating maintenance queries to the sources whose results will refresh the warehouse contents to provide optimal support for user queries.

The IWiz prototype system including the above mentioned WHM components is currently installed and undergoing extensive testing in the Database Research and Development Center at the University of Florida.

CHAPTER 1
INTRODUCTION

A broad spectrum of data is available on the Web in distinct heterogeneous sources and under different formats. Data integration and translation becomes a crucial task for future Web information systems. To provide integrated access to multiple, distributed and independently managed data sources, integration systems need to overcome the discrepancies in the way the data in these sources is maintained, modeled and queried [18, 22].

The main reason for the existence of heterogeneity among the data sources is the use of different hardware and software platforms to store and manage this data. Although the data in the sources may retain some structure, it is not sufficient for conventional data management techniques like *extraction, translation*, and *integration*, to be used effectively [17], due to irregular, unknown or often changing structure of the data. This has sparked great interest in the subject of storing and querying *heterogeneous* and *semistructured* data. To build a tool for extraction of information from semistructured sources it is necessary to account for the various features of semistructured data, as described in the next section.

1.1 Semistructured Data

In recent years, the amount of data available electronically has increased dramatically. Before this, problems associated with storing large amounts of data were solved by using structured databases. Storage formats in these databases (relational or object-oriented) require that all data conform to a predefined schema. Although this limits the variety of data that could be stored, it allows efficient processing of stored data. With the advent of the Web, data that is available to the user now resides in different forms ranging from unstructured data which is stored in file systems (e.g., e-mail, html pages) to highly structured data, which is managed by relational database systems (e.g., product inventory, customer data).

Semistructured data [35] is data that has some structure, but may be irregular and incomplete and does not necessarily conform to a fixed schema as found in standard database systems. The data can be designed with a semistructured format in mind, but generally semistructured data arises as a result of the introduction of some form of structure into unstructured text or when integrating several sources with different schemas. Some of the characteristics of semistructured data are:

- The structure is irregular. Some elements may have missing attributes, others may have multiple occurrences of the same attribute, and the same attribute may have different types in different objects.
- The structure is partial. The degree of structure of data in a document may vary from almost nothing to a fully structured document. Parts of the data may lack structure while some parts may have partial structure.
- An apriori schema can be used to constrain data. This is the basis for traditional database systems where a fixed schema has to be defined before introducing any data. A relaxed approach is to detect a schema from the existing data, to simplify data management, but not for constraining the data.

Querying and storing such irregular, partial data is the focus of many research topics. One such data model used to describe semistructured data and to manipulate this data is the Extensible Markup Language (XML) [41]. XML was introduced as a standard for data representation and exchange on the Web. With the emergence of query languages and persistent storage management techniques for manipulating XML data, XML emerged as a data model.

XML enables the creation of self-describing data structures of arbitrary depth and complexity. One of the main strengths of XML is that it retains the advantages of extensibility, structure and validation required of a language while its design makes it easier to learn, use and implement than other languages. Some of the advantages of using XML are as follows:

- Users can provide new tag and attribute names. XML makes no restrictions on the tags and relationships used to represent the data making it flexible for additional information to be added.
- Document structures can be nested to any level of complexity.
- An XML document can contain an optional description of its grammar for use by applications that need to perform structural validation. These descriptions can be in the form of an XML schema [47, 48] or a Document Type Definition (DTD) [6].

Because of XML's flexibility, portability and simplicity, the industry has started using it as the standard for information interchange. This has fostered intense research and standardization efforts on a data model, query language, typing mechanism etc., specifically designed for XML. One of the query languages for XML documents is XMLQL [11], which is used in our research.

1.2 Motivation and Challenges

The systems used to store and manage XML data can be classified into native XML data stores [19, 34] and relational/object-relational DBMS augmented with an extension to store and manipulate XML Data [30, 51]. From these alternatives, we choose RDBMS for storing and querying XML data because of its high scalability and the fact that a huge set of performance-improving query processing techniques, like *selection*, *extraction*, and *restructuring*, can be exploited for at least certain forms of XML data. Thus using an RDBMS to enable the storage, retrieval and update of XML documents becomes of primary importance as stated in [16, 33, 36].

While using RDBMS as a cache for storing data would be efficient in that querying XML data in the cache would be faster than querying the sources, it raises some issues that need to be resolved. They are as follows:

- If we use a warehouse as a cache for storing data, how do we decide whether to query the sources or the cache?
- If the warehouse cache cannot or can only partially satisfy the user query, we must obtain the desired data from the sources first and then update the contents of the warehouse.
- If we decide the warehouse should be queried to obtain the information, the query must be translated into a format that the warehouse can understand. How do we translate a given XMLQL query into an equivalent SQL query?

While designing solutions to the above-mentioned problems we are faced with three challenges:

1. To allow users to access integrated data from heterogeneous data sources, information can be extracted from the sources or from the cache stored in the warehouse. Querying the warehouse would result in better performance but the data may be absent in the warehouse or if present, may be stale. In either of these situations, the sources must be queried. We should identify the factors that we can use to decide whether to send the queries to the sources or to the warehouse, and use this knowledge effectively.
2. The data warehouse needs to be maintained with up-to-date information so that future user queries can be satisfied with the cached data. Hence *maintenance queries* must be generated so that the data that was previously requested by the user is available in the warehouse. To generate *maintenance queries*, we must find a method to identify what data is missing in the warehouse. Even after we identify what data is to be queried, we have a greater challenge in converting this information to a corresponding set of queries, so the results of these queries update the warehouse.
3. XML uses a hierarchical representation of data, but the structure of the relational schema is flat. One of the main challenges of this research is to translate a given query for XML data which is in semistructured, hierarchical format to a query which expects to conform to a rigid flat schema. With respect to path expressions and relational algebra we need to find out how compatible SQL and XMLQL are, and also detect if there is one-to-one mapping between SQL and XMLQL. These are some of the other challenges faced in query translation.

From this thesis, the reader can expect the following contributions. The first contribution is an algorithm to decide whether the data warehouse can satisfy the requirements of the user query. Second, we develop an algorithm that translates XML queries on the XML views into SQL queries on the data stored in the relational database. And finally, an algorithm to generate maintenance queries based on information that the data in the warehouse is not up-to-date. Updating the contents of the warehouse provides (a) persistence, (b) faster processing of frequently asked queries without having to go to the sources, and (c) automatic maintenance of data contents of the warehouse.

The rest of the thesis is organized as follows. Chapter 2 provides an overview of why we choose XML as our underlying data model and XMLQL as our query language. We also summarize the main research initiatives related to query transformations and warehouse maintenance. Chapter 3 describes the IWiz architecture focusing on the Warehouse Manager and its components as they relate to the work described in this thesis. Chapter 4 focuses on our design and implementation of the decision module, the query translator and the maintenance query generator. Chapter 5 describes the experiments conducted to verify the correctness of results generated by our implementation, and Chapter 6 concludes the thesis with the summary of our accomplishments and issues to be considered in the future.

CHAPTER 2
RELATED RESEARCH

The concept of using XML to represent and manage data is relatively new. We start our description of the related research by summarizing the features of XML and XMLQL, a query language for XML. We then focus on research done on query transformations using the relational database approach for storing and querying XML documents. In the final section, we focus on methods for warehouse maintenance.

2.1 Overview of XML Related Technologies

2.1.1 XML

XML started as a format for representation and data exchange, but was quickly enriched with extensive application programming interfaces (APIs), data definition facilities and presentation mechanisms to make it a suitable data model for semistructured data. XML describes a class of data objects called XML Documents. The structure of an XML document is defined by an optional Document Type Definition (DTD), essentially a grammar for restricting the structure of the document. The DTD can either be included in the document itself, or stored as a DTD file that is referred by the XML document. An XML document satisfying a DTD grammar is considered *valid*. The popularity of XML can be attributed to the following [28]:

- XML is made extensible by allowing new sets of tags to be constructed.
- XML is self-describing. Each data element has a descriptive tag. Using these tags, the document structure can be extracted without knowledge of the domain or the document description.
- XML is able to capture hierarchical information and preserve parent-child relationships between real-world concepts.
- XML allows recursive definitions as well as multiple occurrences of an element.

With the above introduction of XML, we briefly describe the structure of an XML document through an example. Figure 2-1 shows a sample XML document. The line numbers are not part of the XML document, but used only to facilitate our explanation of the example.

The data in XML documents are represented as declarations and elements, which are described as we follow this example. The XML document shown in the figure starts with an XML declaration, which specifies the version of XML being used. The version number *1.0* is used to indicate conformance to this version of the specification. The second line is a comment; therefore, it should not be considered as part of the XML data/schema representation. Line 3 refers to an externally stored DTD file (as explained above) used by this XML Document. Lines 4 to 26 contain the actual data that conforms to the structure defined in the DTD.

Nested tagged elements are the building blocks of XML. A document has a root tagged element that can contain other elements. Each tagged element has a sequence of zero or more attribute/value pairs, and a sequence of zero or more subelements. In our example, in Line 5, the element *book* has an attribute *year* with the value "*1997.*" The subelements may themselves be tagged elements, or they may be "tagless" segments of text data. Lines 6 to 17 in our example containing the elements *author, author*, *title*, and *publisher* are subelements of *book*.

The DTD that defines the structure (schema) for the data being represented is shown in Figure 2-2. An XML document is said to be *valid* if it has a DTD and it conforms to the structure defined in its DTD. To be valid, an XML document also needs to be *well-formed*, which implies that the document and any referenced entity should comply with the XML grammar.

```
<?xml version="1.0"?>
<!-- Sample XML Document -->
<!DOCTYPE document SYSTEM "source1.dtd">
<document>
    <book year = "1997">
        <author>
            <firstname> Joe </firstname>
            <lastname> Bob </lastname>
        </author>
        <author>
            <lastname> Mark </lastname>
        </author>
        <title> Database systems </title>
        <publisher>
            <name> Addison-Wesley </name>
            <address> New York, USA </address>
        </publisher>
     </book>
     <article name = "Computer Hardware">
         <author>
             <lastname> Mathews </lastname>
         </author>
         <title> Reduction of gates in an IC </title>
         <year> 1966 </year>
     </article>
</document>
```

Figure 2-1: Sample XML Document

Elements are the most important logical structures declared in the DTD and may hold single-string typed values (indicated by the keyword #PCDATA), like the element *title* declared in Line 11 of Figure 2-2. Elements may also be nested to represent more complex concepts, like the element *book* in Line 4, which is defined by the child elements *author*, *title*, and *publisher*. Special characters after the element name indicate how many instances of that element can occur under another one. For example, to reflect the fact a book could have one or more authors we use a "+" after the *author* element name in the book declaration (Line 4).

```
 1  <?xml version="1.0"?>
 2  <!DOCTYPE document [
 3  <!ELEMENT document (book | article)* >
 4  <!ELEMENT book (author+,title,publisher)>
 5  <!ATTLIST book year CDATA #IMPLIED>
 6  <!ELEMENT article (author+,title,year?)>
 7  <!ATTLIST article name CDATA #IMPLIED>
 8  <!ELEMENT author (firstname?,lastname)>
 9  <!ELEMENT firstname (#PCDATA)>
10  <!ELEMENT lastname (#PCDATA)>
11  <!ELEMENT title (#PCDATA)>
12  <!ELEMENT publisher (name,address)>
13  <!ELEMENT name (#PCDATA)>
14  <!ELEMENT address (#PCDATA)>
15  <!ELEMENT year (#PCDATA)>
16  ]>
```

Figure 2-2: The DTD for a given XML document

In addition to elements, DTDs can also contain attribute declarations. Attributes associate name-value pairs with elements. They can be used to further describe data represented by elements (in this case they act more like metadata), to establish more data constraints, or simply to provide an alternative to element nesting. In our example, the attribute *year*, is declared for the element *book* in Line 5. The advantage of using a DTD is that it allows any element to become the root element of a concrete document instance. Thus, different XML documents can have different root elements while still conforming to the same DTD and to the underlying global schema.

2.1.2 XMLQL

A considerable amount of information available on the Web today is semistructured. Therefore tools are needed to query, extract, transform, and integrate data from documents [2, 3]. Query languages for semistructured data have been designed and implemented such as Lorel [1, 20] and UnQL [4]. The World Wide Web Consortium (W3C) is currently coordinating the process of creating a query language for XML [14, 43]. Several proposals were submitted to W3C committee including XML-QL, XQuery [49], XQL [9], XSL [42], XPath [45], XMLGL [44], YaTL [7] just to name a few.

According to W3C, some of the requirements an XML query language [46] should satisfy are as follows:

- It must be able to combine related information from different parts of a given document or from multiple documents.
- It must be able to sort query results.
- Queries must be able to transform XML structures and create new XML structures, and
- Queries must be able to perform simple operations on names, such as tests for equality in element names, attribute names, and processing instruction targets and to perform simple operations on combinations of names and data.

Query languages such as Lorel, YaTL, and XML-QL have similarities in their design approach and query capabilities for querying XML data. They can be divided into a match part, corresponding to a query in our sense, and a construct part, that specifies how new documents or data items are constructed using the matches. For our research, we chose XMLQL as our query language. XMLQL is similar in structure to SQL and provides most of the necessary functionalities (i.e., joins and aggregations). Also, database techniques for query optimization, and query rewriting could be extended to XMLQL. In addition, our decision to use XMLQL was influenced by the existence of a robust implementation by AT&T [50].

A sample XMLQL query is given in Figure 2-3. Like SQL's SELECT-FROM-WHERE clause, XMLQL's construct comes in the form of *WHERE-CONSTRUCT* clause. The structure specified in the *WHERE* clause must conform to the structure of the XML document. The *WHERE* clause has two parts. The first part specifies input data, its schema, and location. The second part, which is optional, indicates a set of filters or conditions. The tag elements that specify the input data are bound using "*$*" symbol to distinguish them from string literals. In our example, $y, $p in Line 4, $t in Line 6 are the tag elements. Join conditions and filters can be declared implicitly or explicitly. The string literal “Addison Wesley” in Line 7 is an implicit filter while the condition that year ($y) must be 1996 (in Line 8) is an explicit filter. The *CONSTRUCT* clause identifies the structures of user-defined views. New tags can be created in the resulting document. The tag <*authors*> in Line 12 is one such example.

```
 1  function query() {
 2    WHERE
 3      <document>
 4        <book year=$y>$p</book>
 5      </document> IN “book. xml”,
 6      <title>$t</title> IN $p,
 7      <publisher>"Addison Wesley"</publisher> IN $p,
 8      $y = 1996
 9    CONSTRUCT
10      <book>
11        <title>$t</title>
12        <authors>
13        { WHERE <author>$a</author> IN $p
14          CONSTRUCT $a }
15        </authors>
16      </book>
17  }
```

Figure 2-3: Sample XMLQL query

Looking at the similarity between XMLQL and SQL, it is evident that the WHERE clause specifying the condition in SQL has the same functionality as the *WHERE* clause of XMLQL. The set of tables being queried using the FROM clause is specified in XMLQL using the *IN* construct in the *WHERE* clause. Similar to the AS clause in SQL to rename results, we can create new tags in the *CONSTRUCT* clause. Inside both *WHERE* and *CONSTRUCT* clauses, the schema is declared as the hierarchy of tags, the fundamental component in XML.

XQuery is a specification that describes a new query language for XML and is designed to meet the requirements identified by the W3C XML Query Working Group. It is designed to be a small, easily implementable language in which queries are concise and easily understood. It is also flexible enough to query a broad spectrum of XML information sources, including both databases and documents. XQuery is derived primarily from an XML query language called Quilt, which is described in the next section. XQuery also has borrowed features from other query languages like the regular path expression from XPath and XQL, the notion of binding variables from XMLQL, methods for restructuring data from SQL. Features like data definition facilities for persistent views, function overloading and polymorphic functions, and facilities for updating XML data are not looked into in this specification and will be the focus of future versions of XQuery.

2.2 Query Transformations

Since a major focus of this thesis is on translating queries from one format to another, we give a brief overview of some of the research done in query transformations. We look at how query translations are performed using Quilt [5], a XML query language; and Lore, a research-oriented project at Stanford University.

The primary requirement of an XML query language is to perform queries on the XML representation of data to extract data, to transform data into new XML representations, or to integrate data from multiple heterogeneous data sources. The data can be database data, object data or other traditional data sources. The current XML Query Requirements have no specification regarding directly querying the databases; neither do they define a normative mapping between databases and XML. Hence as there are no specifications for storage and querying XML data, various approaches like using file systems, relational database systems, and object-oriented storage managers have been identified [15, 36]. Though extensive research has been done on the various approaches for storing XML data, less attention is paid to the querying of this data.

2.2.1 Quilt

Quilt is a query language that uses the best ideas from XML-QL, XPath, XQL, YaTL and XSQL; along with some features of SQL and OQL. The conceptual integrity of Quilt comes from the structure of XML, which is based on hierarchy, sequence, and reference. Although Quilt can be used to retrieve data from objects, relational databases, or other non-XML sources, this data must be expressed in an XML view, and Quilt relies solely on the structure of the XML view in its data model.

Manolescu et al., [26] described an approach of using XML queries in relational databases by normalizing the queries expressed in Quilt, translating them on a generic schema and rewriting the queries, using the relational tables as views over the generic schema. A set of translation rules is used for the translation. Since these translation rules are fixed, only a subset of the queries can be used to query the relational database.

Agora [27] is a data integration system that uses XML as the data interface format and Quilt as the query language, where data flow inside the query processor consists of relational tuples. Several servers each owning and sharing data can be considered to be the sources. The shared data is collaborated in answering user queries. Query execution is done on source-specific wrappers that publish the relational data, while query optimization is done on the server where the query is initiated. The wrappers export meta-data, like available access patterns, cost of executing functions, query processing capabilities in terms of arithmetic expressions, joins. The query optimizer takes in this information from the different wrappers and translates it into an execution plan that distributes the work to be done by different wrappers. A Quilt query, expressed in the form of an FLWR (pronounced "flower") expression is constructed from FOR, LET, WHERE, and RETURN clauses. A FLWR expression is used whenever it is necessary to iterate over the elements of a collection. The FOR clause binds variables by iterating over collections of XML nodes; the WHERE clause specifies selection conditions; and the RETURN clause constructs the results. The Quilt query is then translated into a set of correlated, parameterized SQL queries over the relational generic schema.

Quilt can express queries against diverse sources, ranging from documents to relational databases. On relational data, queries have been written involving joins, outer joins, and grouping. The Quilt language attempts to pull together features from several languages that enable it to operate on a broad range of data sources. From XPath and XQL it draws a powerful path expression syntax that can navigate inside a hierarchical document, selecting a set of nodes that satisfy a complex predicate. From XML-QL it draws the notion of bound variables and a versatile syntax that can generate an output document of arbitrary structure.

2.2.2 The Lore Project at Stanford

The aim of the Lore project [29] was to build a complete database management system for semistructured data. It used a data model called the Object Exchange Model (OEM) with its own query language, Lorel, which was based on OQL.

With the emergence of XML, Lore was migrated to be based on a true XML-oriented data model and Lorel was modified accordingly. Query processing is done in Lorel by parsing the input query and preprocessing it to translate it into an OQL-like query. A logical query plan generator then generates a high-level execution plan for the query. The logical query plans can be translated into different physical query plans. Lorel uses a cost-based optimizer to translate the logical query plan into the estimated best physical plan. There are three approaches for query translation/execution strategy. The first approach is the *top-down* approach where the top level bound object is explored and for each object looks for the existence of the nested objects. The second approach is the *bottom-up* approach where the atomic objects are identified first and then work upwards to obtain the bound object. The third approach is a hybrid approach involving the first two approaches. These approaches give rise to different types of execution strategies to evaluate a simple query.

Lore builds and dynamically maintains a *DataGuide* for every database, which is a summary of the current structure of the database and serves some of the functions a schema serves in a traditional DBMS. Since an XML DTD is a set of grammar rules that restrict the form of an XML document, there is a close relationship between DataGuides and DTDs. Lore can use a DTD to build an "approximate" DataGuide. Also, a keyword and proximity search feature is provided with Lorel. The proximity search feature is implemented using a novel indexing technique to scale to very large databases. The prototype is complete with multi-user support, logging and recovery.

2.3 Warehouse Maintenance

From a user perspective, a data warehouse (DW) is a collection of cleansed, integrated, and summarized data, which is available for on-line analytical queries and decision making. From a system perspective, a DW is a database that collects and stores information from multiple data sources. In today’s dynamic environments, it has become necessary to keep DW up-to-date. Active research is being done in the area of supporting DW maintenance under concurrent data updates, while research in the area of supporting DW for schema changes is less.

Users typically perform complex read-only queries on the data. As changes are made to the data at the sources, the warehouse becomes out-of-date. The data is refreshed periodically by *maintenance transactions*, which propagate updates from the sources. In current warehousing systems, maintenance transactions usually are isolated from users' read activity, limiting availability of the warehouse. One approach for a warehouse management scheme is to maintain the warehouse at night, during which time it is not available to the users. Thus, one can maintain consistency without blocking user queries. One can also update the warehouse instantaneously in response to every change at the data source, which is expensive and gives rise to inconsistent results during the same reader session. An update from the data source will change the results a user might see over a sequence of queries. Quass and Widom [31] discuss a possible approach to this problem by maintaining two versions of each tuple at the DW simultaneously so that the reader sessions and the maintenance transactions do not block each other.

Another approach is to take into account that most users do not need the latest data (relative to the warehouse), but would accept *user defined consistency*, where a user specifies a tolerance limit for difference in data between the current data in the warehouse and the data in the sources. Such an approach should support different maintenance strategies for individual views. Thus, views that do not need to be up-to-date can be maintained in a deferred or periodical way. We implement a modified version of this approach by allowing the user to decide the tolerance limit; but when the tolerance limit is approached, the sources are queried; at which time the user queries may be blocked. Whenever data from the sources is loaded into the warehouse, the timestamps for this data are updated. When the user requests some information, the timestamps for the data requested are noted and if the difference with the current time is within the user defined tolerance limit, the data in the warehouse is assumed to satisfy the user query. If not, the data in the warehouse is assumed to be stale and it has to be updated. Thus maintenance transactions are generated to refresh the data in the warehouse.

The selection of the most appropriate maintenance policy is a complex process affected by several dynamic and static system properties. Any approach of DW maintenance must choose an appropriate set of data propagation (or view maintenance) policies to match source characteristics; and must satisfy quality of service attributes for a data warehouse. The main advantage of using a DW is that it allows for faster data access time since the expensive distributed query processing is conducted previously and then cached locally.

2.4 Architectures for Integration Systems

All approaches to data integration are based on a materialized, central data warehouse, or virtual warehouses, a.k.a. mediators. In the data warehousing approach [24], shown in Figure 2-4, the data is integrated from the sources and stored in a centralized repository, *before* users can issue queries against it. This is also commonly referred to as the *eager* or *in-advance* approach to data integration. An excellent overview of the research issues related to the design and implementation of data warehouses is provided [38].

The other well-known architecture for integration systems is based on mediators. This is shown in Figure 2-5. The mediator-based approach to data integration is also known as *lazy* or *on-demand* approach, because the source data is integrated only when the users issue queries to the mediators. The mediators provide the users with an integrated view of the underlying sources, much like the data warehouse; however, since they do not actually store the corresponding data, the incoming query is rewritten into one or more queries against the sources that participate in the answer.

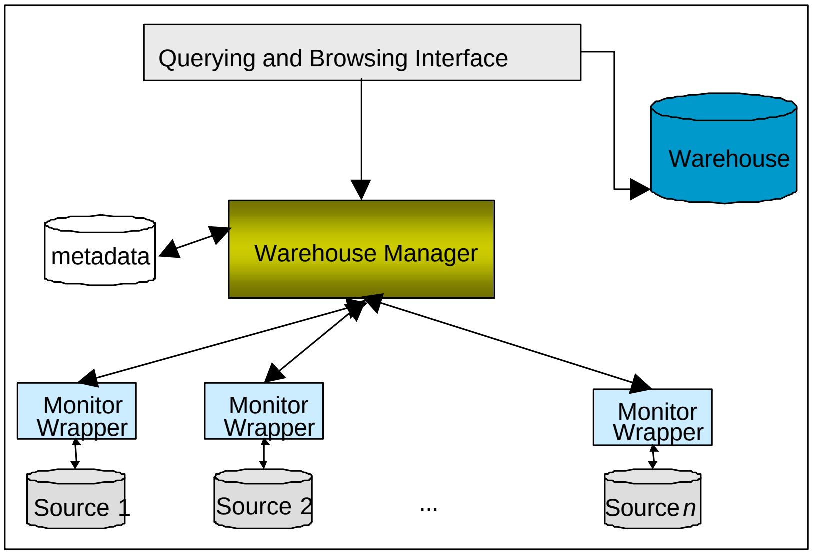

Figure 2-4: Warehouse-based architecture for data integration

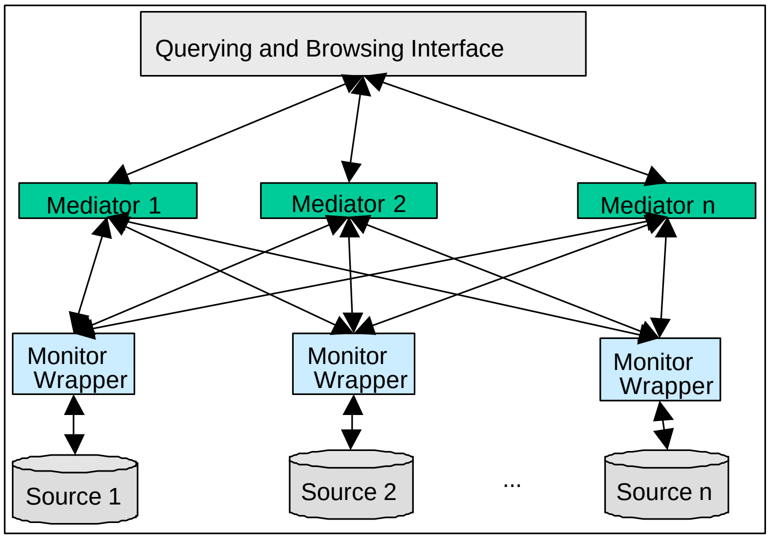

Figure 2-5: Mediator-based architecture for data integration

The Integration Wizard (IWiz) system, under development at the University of Florida, allows users to access and retrieve information from multiple sources through a consistent, integrated view. To improve query response time, we have an underlying relational database to store the data from the different sources, and an interface to query this data. IWiz uses a "hybrid" approach where we have a mediator system with a data cache (relational database) for storing the answers to frequently asked queries. All queries are checked to see if they can be satisfied by the data cache. If so, the data cache is queried and the result returned to the user (*data warehousing approach*). If the cache cannot satisfy the user query, the query is sent to the sources (*mediator approach*) and the merged result from the sources is returned to the user.

CHAPTER 3
THE INTEGRATION WIZARD PROJECT

As pointed out in Chapter 1, this research is part of a bigger effort to develop a new integration system called the Integration Wizard (IWiz). The goal of IWiz is to query and manage interesting data stored in multiple, heterogeneous sources. It focuses on sources containing semistructured data. IWiz aims to provide integrated access to heterogeneous data through one common interface and user-definable view of the integrated data and to warehouse frequently accessed data in integrated fashion for faster retrieval. It provides these functionalities by (a) helping users in describing the desired information in a format suitable to their needs, (b) resolving semantic heterogeneity by automatically restructuring and transforming the relevant source data into a unified data model, and (c) supporting the querying of source data through the user-defined view and transferring the data into the view definition with all the inconsistencies resolved.

3.1 IWiz Architecture

The architecture of IWiz is presented in Figure 3-1 The main components of the systems are the Querying and Browsing Interface, the Warehouse Manager, Mediator and the Wrappers, also called as the Data Restructuring Engine.

In IWiz, data is represented as XML documents. The sources provide the requested data to be integrated. The wrappers translate and restructure the source data and provide them to the mediator. The mediator produces the mediated data by fusing the restructured data and cleaning redundant and overlapping data. This mediated data is the result of a given user query. We shall now provide a brief description of the modules of IWiz [21].

The sources, shown in Figure 3-1, are present at the lowest tier of the architecture. Ideally the sources can be structured, unstructured or semistructured. However, in our current version, the sources are XML based, semistructured sources. These underlying source information is represented using XML together with a DTD which explains the schema of the data.

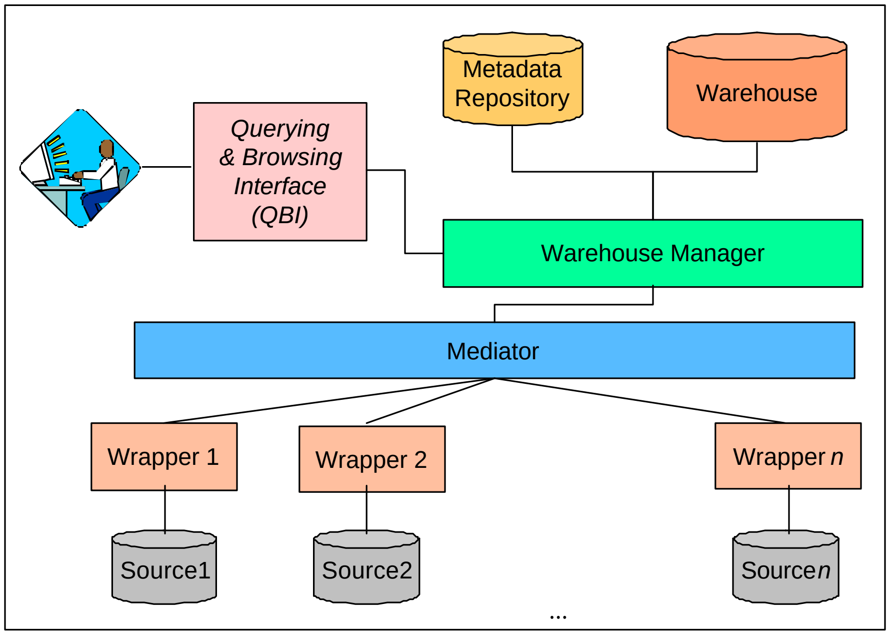

Figure 3-1: IWiz architecture

The Warehouse, shown above the Warehouse Manager in the Figure, is a repository that is used to store user-query results. It provides faster access to frequently asked queries. Oracle 8*i* is the relational database that we are using to store the data. There is also a Metadata Repository that serves as a persistent repository for storing auxiliary data such as global schema, restructuring specifications, merging specifications, information about sources etc., The Querying and Browsing Interface (QBI) provides an interactive graphical interface to create queries. It also lets users view query result in an easy-to-understand and easy-to-explore fashion. The Warehouse Manager (WHM) accepts a user query from the QBI and analyzes whether the query can be satisfied by the warehouse. It returns the result that can be either found in the warehouse or acquired from the mediator to the users via the QBI. It also keeps the data in the warehouse up-to-date by generating maintenance queries and sending it to the Mediator. As shown in Figure 3-1, WHM is connected to QBI, Metadata Repository, Warehouse and the Mediator.

The Mediator which forms the middle tier of the architecture maintains metadata and information about underlying sources. The work of the Mediator is (a) to analyze a query coming from the Warehouse Manager, transform the query into a set of source specific queries and send them to the corresponding Wrappers and (b) to merge the restructured data obtained from the Wrappers and return them to the Warehouse. The Mediator connects to the Warehouse Manager in the top tier and the Wrapper in the third tier as shown in Figure 3-1.

The Wrapper serves as an interpreter between the sources and the rest of the system. Its tasks are to (a) map the data model in the underlying sources into a common data model used in the integration system, (b) map the schema of the underlying sources into semantically equivalent concepts defined in the global schema, (c) transform the mediated query into a restructured query to access the underlying source, and (d) restructure the underlying source data.

Now we give a brief overview of the Warehouse Manager and the three components that are the focus of this research.

3.2 Warehouse Manager

The Warehouse Manager (WHM), shown in Figure 3-2, is responsible for caching the results of frequently accessed queries for faster response and increased efficiency. The XMLQL-2-SQL Query Translator, the Decision Module and the Maintenance Query Generator which are the research areas of this thesis are components of WHM.

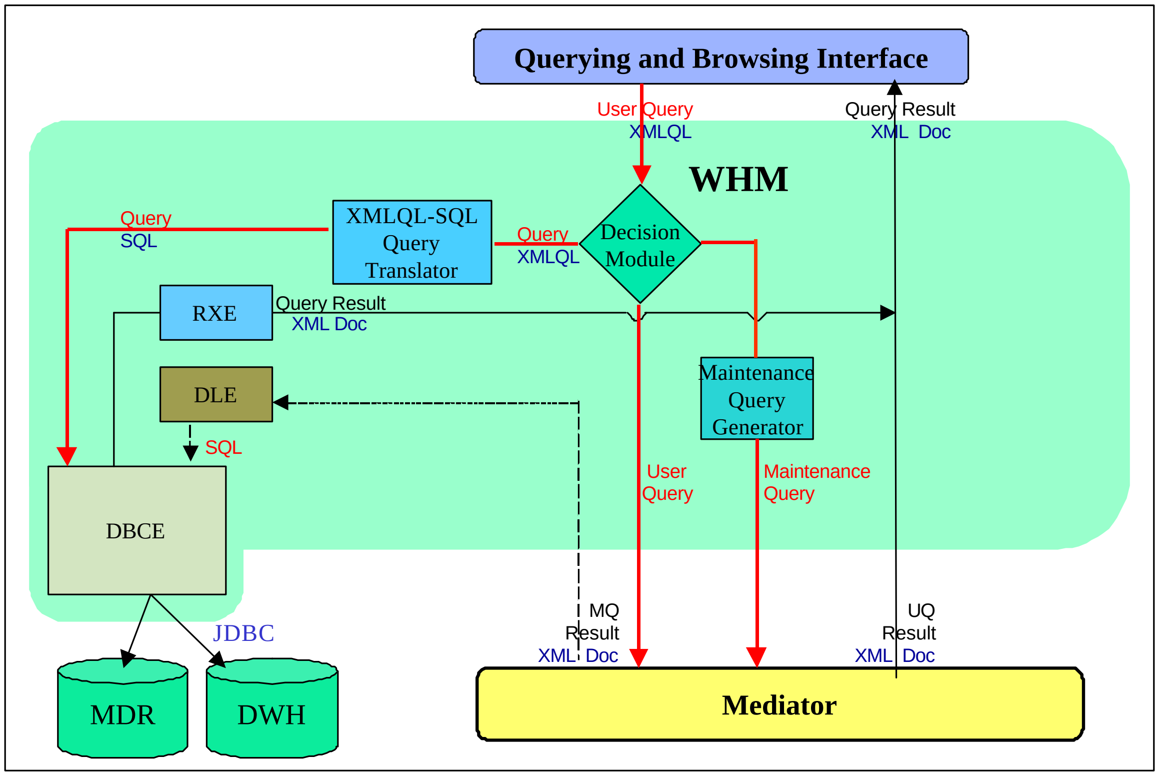

Figure 3-2: WHM architecture

In Figure 3-2 shown above, the Querying and Browsing Interface is the interface with which the users interact to obtain results to their queries. The user query (UQ as shown in the figure) which is in XMLQL is sent to the Warehouse Manager (WHM). The Decision Module decides whether the warehouse (DWH) can satisfy the user query. If so, it is translated to a SQL query by the XMLQL-to-SQL query translator and sent to DWH through the Database Connectivity Engine (DBCE). The result set is then converted to an

XML document by the Relational-to-XML Engine (RXE) and given to the user. If the DWH cannot satisfy the user query, it is sent to the Mediator. The resulting XML document (UQ Result) is sent to the user. Also, Maintenance Queries (MQ) are generated which are also sent to the Mediator. The resulting XML document (MQ Result) is loaded into the DWH by the Data Loader Engine (DLE). In the figure, the dark lines indicate the flow of queries (XMLQL and SQL) while the light and dotted lines indicate the flow of XML documents. The dotted lines represent the flow of data to the DWH while the light lines indicate the flow of data to the user.

Before we discuss the architecture and implementation of the work presented in this thesis, we describe the features related to WHM that are important for a better understanding of our thesis. The two major phases in the WHM operation are the *built-time phase* and *the run-time phase*. At *built-time*, the DTD description of the global IWiz schema is processed to generate the relational schema for the warehouse. At *run-time*, WHM accepts and processes user queries which are in XMLQL format. When the XMLQL query is provided to WHM, the Decision Module determines whether the contents of the warehouse can satisfy the user query. The XMLQL-2-SQL Query Translator generates an equivalent SQL query for the given XMLQL query. This SQL query is executed against the warehouse. The relational result generated is then converted to an XML document by the Relational-to-XML-Engine (RXE). If the Decision Module decides that the warehouse cannot satisfy the user query, it sends the user query to the Mediator. The resulting XML document generated by merging the information from various sources after resolving conflicts is sent to the Querying and Browsing Interface to be displayed to the user. To keep the contents of the warehouse up-to-date, the

Maintenance Query Generator generates a maintenance query which is sent to the Mediator. The resulting XML document is loaded into the warehouse by the XML Loader (DLE) component of WHM. The RXE and DLE are part of another ongoing research project in the Database Research and Development Center at the University of Florida.

3.3 Overview of Relational Approach for Managing XML Data

There have been numerous studies for storing XML documents and for executing queries on that data [17]. XML storage strategies can be classified into three categories according to the underlying system used: file system, storage manager or database management systems (DBMS). A storage manager can be a file system augmented with database features or a database system with file-system features. We can store XML documents as ASCII files. The disadvantage of this approach is that they need to be parsed every time they are accessed. Also the merging and updating operations are difficult to implement. The second approach, using the storage manager, involves using an object manager. While this can be expected to provide better performance, the record-level interface for querying provided by a typical storage manager requires more work to use than a query language using a database system.

Regarding the kind of DBMS used for integration and querying of XML documents, one can identify four different approaches. First, special-purpose DBMS are particularly tailored to store, retrieve, and update XML documents. Lore, Strudel [13] are examples of some of the research prototypes. Second, because of the rich data modeling capabilities of object-oriented DBMS, they are well-suited for storing hyper-text documents. Third, object-relational DBMS could be used since the nested structure of the object-relational model blends well with XML's nested document model. However the above-mentioned approaches have not been explored thoroughly to handle large scale data in an efficient way and also they are not in wide-spread use.

The fourth and final approach and the one that we have selected, is to store XML documents in a relational database management system (RDBMS). Advantages such as reusing a mature technology and seamlessly querying data represented in XML documents influenced our decision. Within an RDBMS, we have three alternatives for storage. The easiest approach would be to store the XML documents as a whole within a single database attribute. The second would be to interpret XML documents as graph structures and provide a relational schema allowing to store arbitrary graph structures. The third approach is that the structure of an XML document e.g., DTD, is mapped to a corresponding relational schema wherein XML documents are stored according to the mapping. As this allows us to exploit the features of RDBMS such as querying, optimization, concurrency control etc., we have used this approach in our implementation of WHM in IWiz.

Using RDBMS to store XML documents raises the issue of how to query these documents. One could ask why not adapt SQL to query XML. SQL cannot be modified to query XML because XML data is fundamentally different from relational data. In relational data models, every data instance has a schema which is separate from and independent of the data, while in XML the schema exists with the data. Also, in XML, data items may have missing elements or multiple occurrences of the same element; elements may have atomic values in some data items and structured values in others; and collections of items can have heterogeneous structure. Hence we need a mechanism to translate XML queries into SQL.

3.4 Architecture Overview

We shall now give a brief overview of the architecture which is divided into the *built-time* and *run-time* phase.

3.4.1 Built-time Phase

The *built-time* architecture of WHM is shown in Figure 3-3. During this phase, only the Decision Module is used. The input to this module is the Maintenance Key File. It contains a list of the variables for which a maintenance query must be generated. A hashtable, *Maintenance hashtable,* is created from information in the input, which is also stored persistently in the warehouse. Metadata from the Meta Data Repository is used to create (1) the *Tag Info hashtable* which identifies whether the variables in the global schema are CONCEPTs or FIELDs/ATTRIBUTEs, (2) *Table Info hashtable* which stores the names of all columns associated with a table in the relational schema, (3) *Associated Table Info hashtable* which contains the names of all tables that are mapped to a table in the relational schema, and (4) the *Time Stamp Info hashtable* which contains information regarding the freshness of the data in the warehouse. These terms are explained in detail in the next chapter. The hashtables created are stored together in a module called DataContainer, that can be accessed by the various modules of WHM. All these are also stored persistently in the warehouse. While the first three are fixed for a given relational schema, the *Time Stamp Info hashtable* may be modified based on the changes to the data in the warehouse.

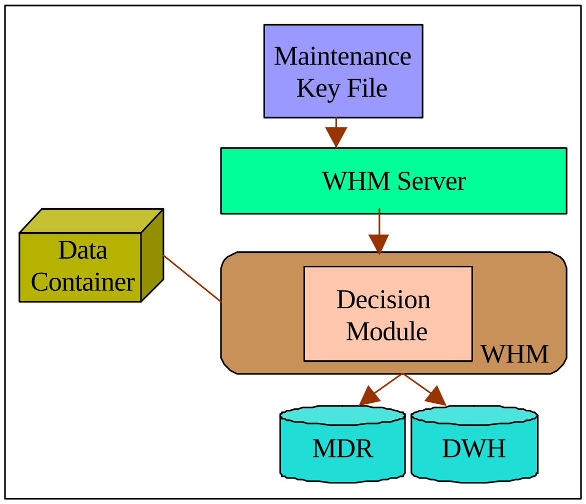

Figure 3-3: Built-time architecture

3.4.2 Run-time Phase

In the *run-time* phase, the Decision Module (DM), the XMLQL-2-SQL Query Translator (QT), and the Maintenance Query Generator (MQG) are used. The *run-time* architecture is shown in Figure 3-4. The modules are aided by the Data Container (DC) which is used to store the metadata required for processing the user queries. At *run-time*, WHM is ready to accept user queries from the Query and Browsing Interface (QBI), as described. These user queries are in XMLQL format. When the XMLQL query is provided to the WHM, the DM determines whether the contents of the warehouse can satisfy the user query. The QT then translates the given XMLQL query into an equivalent SQL query which is executed against the warehouse. The relational result generated is then converted to an XML document that is returned as the result to the user query back to the QBI. The QBI then displays the result to the user. If the DM decides that the warehouse cannot satisfy the user query, it sends the user query to the Mediator component. The resulting XML document generated by merging the information from various sources after resolving conflicts is sent to the Querying and Browsing Interface to be displayed to the user. To keep the contents of the warehouse up-to-date, the MQG generates a maintenance query which is sent to the mediator.

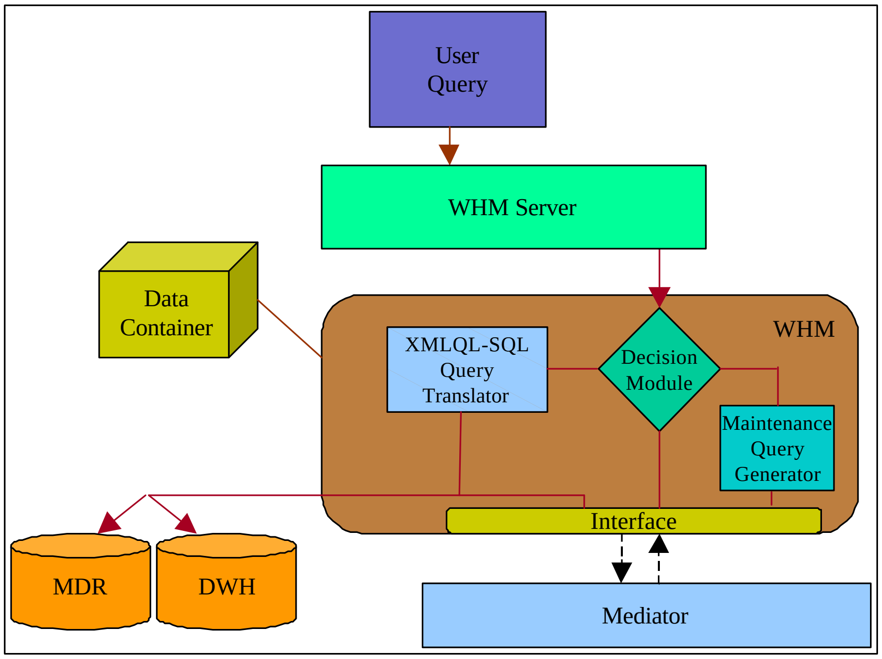

Figure 3-4: Run-time architecture

The maintenance query generates results that are used to load the data warehouse so that in the future, similar queries can be satisfied directly from the warehouse. The resulting XML document is loaded into the warehouse.

We shall now explain the sample application used in our implementation that is important for a better understanding of our thesis. To illustrate our approach, we describe a scenario where users would like to get information pertaining to description and identification of the editions, dates of issue, authorship, and typography etc., of books or other written material. This descriptive information is termed as bibliography, on which the global schema (DTD) of IWiz is based. The structure of the DTD corresponds to the concept of a bibliography. The global (integrated) schema includes most of the commonly used items of a bibliography and their descriptions. A subset of the DTD is shown in Figure 3-5. The detailed DTD is included in the Appendix.

```
<!ELEMENT Ontology (Bib*)>
<!ELEMENT Bib(Book,Article)*>
<!ELEMENT Book (Author+, Title, Year, Editor*, ISBN>
<!ELEMENT Article (Author*, Title, Year, Editor?>
<!ELEMENT Author (Firstname?, Lastname, Address>
<!ELEMENT Editor (Lastname)>
<!ELEMENT Title (#PCDATA)>
<!ELEMENT ISBN (#PCDATA)>
<!ELEMENT Year (#PCDATA)>
<!ELEMENT Firstname (#PCDATA)>
<!ELEMENT Lastname (#PCDATA)>
<!ELEMENT Address (#PCDATA)>
```

Figure 3-5: A subset of the DTD associated with the application

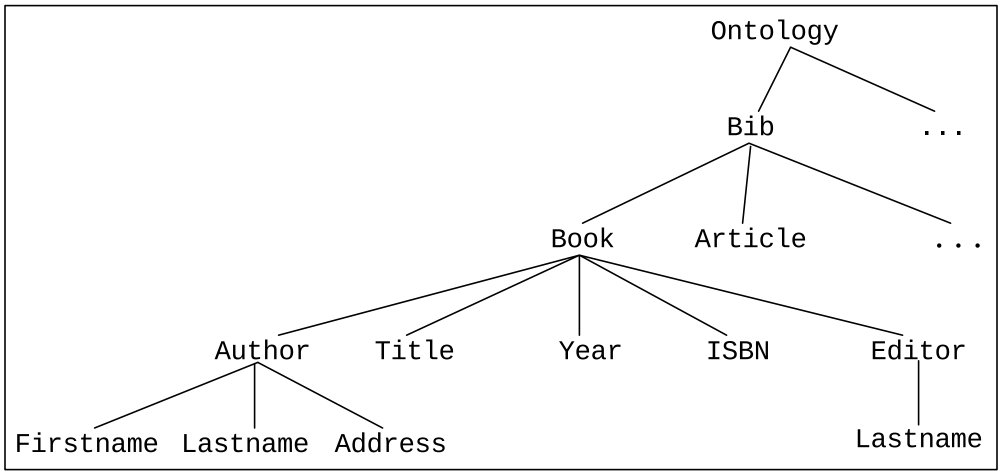

Figure 3-6: Hierarchical structure of the global IWiz schema

The root of the global schema is the element *Ontology*. An ontology can contain one or more types of bibliographies specified by the element *Bib*. Each bibliography has multiple instances of *Book*s and *Article*s. Each instance of *Book* can contain one or more *Author*s, a *Title*, an *ISBN*, the *Year* it was published and zero or more *Editor*s. Similarly each instance of article has zero or more *Author*s, the *Title*, the *Year* it was published and one or more *Editor*s. Finally each author can have an optional *Firstname*, a *Lastname* and an *Address* while an editor has only a *Lastname*.

The hierarchical structure of the schema is shown in Figure 3-6. It contains the same information as Figure 3-5, but in a tree structure for easier understanding of the DTD. The root of the tree is *Ontology,* which has as its children a set of bibliographies, *Bib*. Each *Bib* in turn contains one or more *Book*s and *Author*s. This structure continues till we reach the leaf nodes of the tree, which are the elements with PCDATA in our DTD. Thus all non-leaf nodes are elements in the DTD which contain other elements as children while the leaf nodes must be either elements having PCDATA or must be attributes (as explained in Line 5 of Figure 2-2).

CHAPTER 4
DESIGN AND IMPLEMENTATION

Before describing the design and implementation of the Decision Module, the XMLQL-2-SQL Query Translator and the Maintenance Query Generator we first introduce some important terms and data structures used in the reminder of this chapter:

- *Concept*: In DTD terms, a CONCEPT is an element that satisfies one or more of the following conditions (a) it has one or more attributes, (b) it has one or more children, or (c) can occur more than one time. In relational terms, a CONCEPT is a table. In our sample schema, shown in Figure 3-5, *Book*, *Article*, *Author*, *Editor* are all concepts.
- *Field/Attribute*: In a DTD, a FIELD is an element which has no children. It is the leaf node in a hierarchical structure of the DTD. FIELDs are mapped to columns of a relational table. In Figure 3-6, which is the hierarchical structure of the schema of Figure 3-5, *Firstname*, *Lastname*, *Title*, and *Year* are all FIELDs/ATTRIBUTEs.
- *parentChild*: parentChild is a term which we shall be using frequently in the remainder of this chapter. Referring to Figure 3-6, parentChild refers to a node in the tree and its parent. Thus, Bib with either Book or with Article would refer to a parentChild. We write this in the form of *parent* followed by a ”:” and then the *child*. Thus, *Ontology:Bib*, *Bib:Book*, *Author:Lastname*, *Book:Year* are all parentChild elements. One can notice that the parentChild element can refer to a CONCEPT, FIELD or ATTRIBUTE. The same idea is expressed in relational terms as *tableColumn* written as *table*:*Column*. But only those elements which are FIELDs or ATTRIBUTEs occur as *tableColumn* elements.
- *Maintenance hashtable*: A maintenance hashtable is a data structure that is given as input at built-time. This is shown in Figure 4-1.
  This is used by the Maintenance Query Generator module to generate maintenance queries. The data structure comprises of a set of elements for which maintenance queries must be generated and the path of these elements from the root in the global schema. For example, the element *Book* in Figure 3-6 has the path Ontology\Bib\Book from the root, which is the value for the key ”Book” in the Maintenance hashtable.

Key	Value
Book	Ontology.Bib.Book
Article	Ontology.Bib.Article

Figure 4-1: Maintenance hashtable given as input at built-time

Key	Value
Bib:Article	CONCEPT
Ontology:Bib	CONCEPT
Author:Lastname	INLINED-FIELD
Article:Title	INLINED-FIELD
Article:Editor	CONCEPT
Book:Editor	CONCEPT

Figure 4-2: Tag Info hashtable

- *Tag Info hashtable*: The Tag Info hashtable, shown in Figure 4-2, is created at built-time. The keys are the parentChild (explained above) and the values indicate whether these parentChild elements are CONCEPTs, FIELDs or ATTRIBUTEs. Thus, *Article*, which is a CONCEPT in Figure 3-5, is shown with the key *Bib:Article* and the value *CONCEPT*, while *Lastname* which is an element of *Author* (*Author:Lastname)* has the value *FIELD*.
- *Table Info hashtable*: This data structure is based on the relational schema created from the global schema. As shown in Figure 4-3, this data structure consists of a set of names of the tables created in the relational schema and the columns of the table.

Key	Value
Bib	[BIB_PK_ID]
Ontology	[ONTOLOGY_PK_ID]
Author	[AUTHOR_PK_ID, FIRSTNAME, LASTNAME, ADDRESS]
Editor	[EDITOR_PK_ID, LASTNAME]
Article	[ARTICLE_PK_ID, TITLE, YEAR]

Figure 4-3: Table Info hashtable, based on the relational schema

As shown in Figure 4-3, each table has a primary key associated with the table, which is not present in the global schema (Figure 3-5).

- *Associated Table Info hashtable*: This data structure, shown in Figure 4-4, is also created based on the relational schema. Referring to Figure 3-6, each node in the tree can have as its child, a CONCEPT or a FIELD/ATTRIBUTE. Those non-leaf nodes in the tree whose children are non-leaf nodes form the keys of this hashtable and the children of these nodes, which are CONCEPTS themselves, are the values.

Key	Value
Bib	[Article, Book]
Book	[Author, Editor]
Article	[Author, Editor]
Ontology	[Bib]

Figure 4-4: Associated Table Info hashtable

As can be seen, Ontology is a non-leaf node which has Bib as its child and this is reflected in Figure 4-4. Similarly, it can be seen that the associated tables of Book are Author and Editor, and that Author is not be present in this hashtable as a key element as it has no non-leaf nodes.

- *Time Stamp Info hashtable*: The *Time Stamp Info hashtable*, shown in Figure 4-5, is created at *built-time*. The parentChild terminology explained above are the keys of this hashtable. They refer to the columns of the tables created in the relational schema. The value of each key is the time when the warehouse was updated with respect to this table and particular column. Thus, the value for the key *BOOK:YEAR* refers to the latest update on the Month column of the Article Table. At built-time, when the warehouse is initially empty, there are no values for any of the keys.

Key	Value
BOOK:YEAR	2001-03-25 16:17:25.0
ARTICLE:AUTHOR	2001-03-25 16:17:25.0
AUTHOR:LASTNAME	2001-03-25 16:17:25.0
ONTOLOGY:BIB	2001-03-25 16:17:25.0

Figure 4-5: Time Stamp Info hashtable

A detailed structure of the DTD, *Maintenance hashtable*, *Tag Info hashtable*, *Table Info hashtable*, *Associated Table Info hashtable* and *Time Stamp Info hashtable* is presented in the Appendix.

With this understanding of interaction between the various components of WHM, we now proceed to describe in detail the design and implementation of the XMLQL-2-SQL Query Translator, the Decision Module, and the Maintenance Query Generator components that are the focus of this research.

4.1 Decision Module (DM)

As explained in Chapter 2, IWiz uses a "hybrid" approach as the approach for data integration, in which we have a mediator system with a data cache for storing the answers to frequently asked queries. Because of this, a decision has to be made to either query the cache or send the query to the sources. The DM in WHM performs this function. When a query is submitted by the QBI, the DM parses the incoming XMLQL query and obtains the set of tags that are queried and requested by the user. Both the WHERE clause and the CONSTRUCT clause of the XMLQL query may contain these tags (bound variables).

```
 1 Parse the incoming query to obtain the tags that need to be queried
        in the warehouse
 2 For each tag
 3   Obtain the time when the warehouse was last modified w.r.t this tag
 4   If the difference between the current time and the timestamp of the
         tag in the warehouse is beyond a threshold limit, it means the
         query cannot be satisfied by the warehouse
 5 If the query cannot be satisfied by the warehouse
 6   Query the sources through the Mediator
 7   The resultant XML document is sent to the QBI
 8   Generate a set of XMLQL queries so that the warehouse can be updated
 9   Update the timestamp of those tags in the warehouse whose contents
         are updated
10 If the query can be satisfied by the warehouse
11   Call the XMLQL-SQL Query Translator to translate the XMLQL query to
         the corresponding SQL query in the underlying relational schema
12   The SQL query generated is then queried on the data warehouse and
         the resultant XML document generated is sent back to the user
```

Figure 4-6: Pseudo code of the Decision Module algorithm

At *built-time*, the DM connects to the database to get information about the timestamps of the variables present in the global schema. The pseudo code of the Decision Module Algorithm is given in Figure 4-6. We shall explain the pseudo code in detail.

The incoming query is parsed to obtain the set of tags that participate in the query. This is shown in Line 1. The tags can be present in the WHERE clause as bound variables or conditions/filters. For each queried tag, the timestamps is checked with the current time (Lines 2 and 3). If the difference in the time is within a threshold as determined at *built-time*, the contents of the warehouse for those variables queried are considered to be fresh in the warehouse (Line 10). If so, it calls the XMLQL-2-SQL Translator to translate the XMLQL query into a SQL query to query the warehouse (Line

11). The result set of the SQL query is converted to an XML document and sent to the user via the Querying and Browsing Interface as indicated by Line 12. If the warehouse cannot satisfy the user query (Line 4), it has to query the sources. The user query is then sent to the mediator (Line 6). The result of the user query obtained from the Mediator is sent back to the user as explained in Line 7. At the same time, the DM calls the Maintenance Query Generator, to generate a set of maintenance queries to update the warehouse (Line 8), while the result of the maintenance query from the Mediator is used to update the contents of the warehouse. The timestamps that are used to indicate the freshness of data in the warehouse is updated for those variables whose data is loaded into the warehouse. This operation is explained in Line 9 of the pseudo code. We shall now describe the implementation of the XMLQL-to-SQL Translator.

4.2 XMLQL-2-SQL Query Translator (QT)

If the DM decides that the contents of the warehouse can be used to satisfy the user query, it calls the QT, which translates the user query to a SQL query to query the warehouse. The design issues and implementation of QT is discussed in this section.

Consider a scenario in which the user likes to get information about books and articles whose authors have the same lastnames. Specifically, the user wants only the titles of books and articles, which have been written by authors having the same lastname and have been published in the year 1995. The query generated by QBI would in the form shown in Figure 4-7.

```
function query() {
  WHERE
    <document>
       <book year=$year>
          <title> $title_book </title>
          <author>
             <lastname>
                <PCDATA> $author_book </PCDATA>
             </lastname>
          </author>
       </book>
       <article>
          <title> $title_article </title>
          <author>
             <lastname>
                <PCDATA> $author_article </PCDATA>
             </lastname>
          </author>
          <year> $year </year>
        </article>
    </document> IN IWiz,
     $year = 1995,
     $author_book = $author_article
  CONSTRUCT
     <authors_of_book_or_article year=$year>
        $author_book
        <book_title>$title_book </book_title>
        <article_title>$title_article</article_title>
     </authors_of_book_or_article>
}
```

Figure 4-7: An XMLQL query requesting information about books and articles

This query conforms to the DTD shown in Figure 2-2. Before explaining in detail how the Query Translator algorithm works, we shall explain some terms used in the algorithm: the XMLQL query is parsed to obtain those tags that are queried in the XMLQL query. Both the bound variable and the path of this element from the root are stored in a hashtable. We shall call this hashtable as *Queried Tag hashtable*. The structure of the hashtable is shown in Figure 4-8. This hashtable is created for the query given in Figure 4-7. As can be seen from Figure 4-7, *$title_article* is a variable bound to the title of a book. Hence this is the value in Figure 4-8, which has as its key, the path from the root of the global schema (i.e., document).

Key	Value
\document\article\title	$title_article
\document\article\author\lastname	$author_article
\document\book\title	$title_book
\document\book\author\lastname	$author_book
\document\book\year	$year
\document\article\year	$year

Figure 4-8 Queried tag hashtable created for the XMLQL query of Figure 4-7

Also, at the same time, the conditions present in the XMLQL query are noted. We shall refer to the storage data structure of these conditions as the *ConditionInfoVector*. The structure of the *ConditionInfoVector* for the given query is shown in Figure 4-9.

```
$year        = 1995
$author_book = $author_article
```

Figure 4-9: *ConditionInfoVector* created for the XMLQL query of Figure 4-7

This data structure informs us that the year in which both book and article were published must be “1995.” Also, the second condition is that the lastname of the author of a book must be the same as the lastname of the author of an article. In general, all the conditions and filters specified in a query are stored in the *ConditionInfoVector*. This includes implicit filters (e.g., <year>“1995”</year>), explicit filters (e.g., $year = 1995), and joins conditions (e.g., $author_book = $author_article).

The pseudocode of the Query Translation algorithm is shown in Figure 4-10. Let us explain the algorithm in detail. Line numbers are present to facilitate easier understanding of the pseudo code. After creating the *Queried Tag hashtable,* each tag is processed (Line 1). For each tag, as indicated in Lines 2 and 3, the parentChild is obtained (explained earlier in this chapter). The parentChild refers to the bound variable and its parent in the hierarchical structure of the global schema. Thus, for example, for the bound variable *$title_book*, the parentChild tag would be *book:title*, and for *$article_author*, the parentChild would be *author:lastname*. After obtaining the parentChild for a tag, find out whether this bound element refers to a CONCEPT or a FIELD/ATTRIBUTE by checking the *Tag Info hashtable*.

```
 1 for all keys in the QueriedTagHashtable
 2   for each key
 3     get the parentChild name of the key
 4     if key is CONCEPT
 5       if it is queried in the construct clause
 6         if the path of the key already contains a TABLE, it is an ERROR
 7         select the table with the current table index number and increase the
                 table index number
 8         from the Table Info Hashtable obtain the columns of the table and insert
                  in the hashtable the column names and the keyword "SQL" with
                  the current column index number and increase the column index number
 9         process the same algorithm for all associated tables of this table
10     if key is FIELD or ATTRIBUTE
11       if the path of the key already contains a TABLE
12         if it is queried in the construct clause
13           obtain the vector which contains the column names and its
                   associated sql names
14           add to the vector the last string from '\' as the column name and
                   the keyword "SQL" with the current column index number
15           increase the column index number
16       else
17         select the table with the current table index number and increase the
                  table index number
18         insert in the hashtable the column name and the keyword "SQL"
                  with the current column index number
19         increase the column index number
```

Figure 4-10: Pseudo code of the SQL Query Generation algorithm

If the bound element is a CONCEPT (Line 4), and it is queried in the CONSTRUCT clause (Line 5), check if the TABLE has been selected for this CONCEPT in the SQL query. Note that a CONCEPT in XML terms is the same as a TABLE in relational terms. If this table has been selected, then there is an error in the query and the user is informed about this. This is indicated in Line 6. Also, note that a CONCEPT cannot be associated with a condition, as it has many child elements and each child element must be associated with the condition individually. If no table has been selected, then this table is selected with an alias (Line 7). Since it is a CONCEPT, all the columns of this table must also be selected in the SQL query. Hence, from the *Table Info hashtable*, select all the columns associated with this table. The primary key of the table is ignored, as it is not present in the global schema. This is explained in Line 8. Also, this CONCEPT can have as its children other CONCEPTs. Hence the algorithm must recursively process the tables associated with this table by obtaining the information from *Associated Table Info hashtable*. Thus the same algorithm is called for all tables that are associated with this table as shown in Line 9.

If the bound element is a FIELD/ATTRIBUTE (Line 10), it can be present either in the CONSTRUCT clause or in the *ConditionInfoVector*. If the variable is present only in the *ConditionInfoVector* then in the SQL query generated, this variable should be present only in the *WHERE* clause. If it is present in the CONSTRUCT clause, this variable shall be selected in the *SELECT* clause. Check if the TABLE has been selected which is associated with this FIELD/ATTRIBUTE, as explained in Line 11. If the TABLE has already been selected, then obtain the set of those columns selected for this TABLE and append the column associated with this bound element to that set. This is explained in Lines 13 to 15. If the TABLE has not been selected, then select this TABLE and the associated column. These steps are explained in Lines 17 to 19.

The above-explained process is executed recursively for all the bound elements present in the *Queried Tag hashtable*.

Looking at the query, one can notice that authors of both books and articles are queried. In the relational schema designed from the global XML schema, it can be seen that there is only one author table created. Data in this table refers to authors of book and articles. Hence we must be able to identify those tuples which are associated with books and those tuples which are associated with articles. This issue is resolved in our implementation by the creation of *Relational mapping tables*. The *Relational mapping tables* maps two given tables with their primary keys. For example, since there is a mapping to the author table from both book table and article table, two mapping tables “book_author” and “article_author” are created. For the given XMLQL query, the relational mapping tables are shown in Figure 4-11.

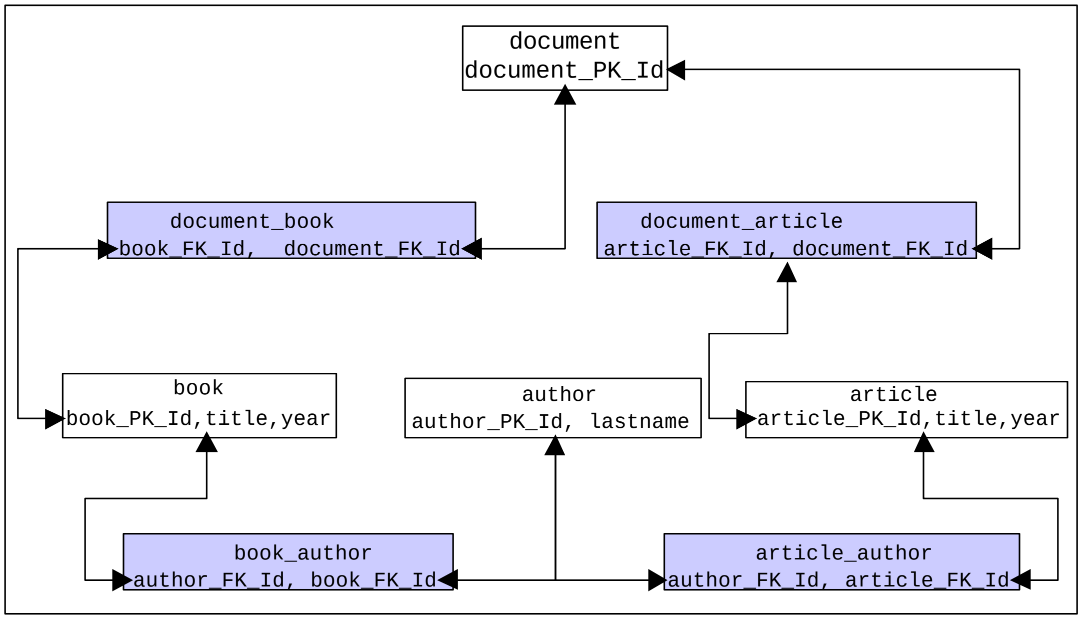

Figure 4-11: Relational mapping tables for the given XMLQL query

Also, one must taken into consideration grouping associated with an XMLQL query. This can be taken care in the corresponding SQL query with a *group by* clause. Hence those bound elements which are requested in a *group by* form in the XMLQL query is first selected in the *group by* clause of the SQL and then the remaining variables in the *Select* clause are added.

```
Select
       TABLE6.TITLE As sql2,
       TABLE8.LASTNAME As sql3,
       TABLE1.TITLE As sql4,
       TABLE1.YEAR As sql1
From
       ARTICLE TABLE1,
       ARTICLE_AUTHOR TABLE5,
       BOOK_AUTHOR TABLE9,
       DOCUMENT TABLE2,
       DOCUMENT_BOOK TABLE7,
       AUTHOR TABLE8,
       AUTHOR TABLE4,
       BOOK TABLE6,
       DOCUMENT_ARTICLE TABLE3
Where
       TABLE1.YEAR LIKE '%1995%'
        AND TABLE8.LASTNAME LIKE TABLE4.LASTNAME
        AND TABLE2.DOCUMENT_PK_ID = TABLE3.DOCUMENT_FK_ID
        AND TABLE3.ARTICLE_FK_ID = TABLE1.ARTICLE_PK_ID
        AND TABLE1.ARTICLE_PK_ID = TABLE5.ARTICLE_FK_ID
        AND TABLE5.AUTHOR_FK_ID = TABLE4.AUTHOR_PK_ID
        AND TABLE2.DOCUMENT_PK_ID = TABLE7.DOCUMENT_FK_ID
        AND TABLE7.BOOK_FK_ID = TABLE6.BOOK_PK_ID
        AND TABLE6.BOOK_PK_ID = TABLE9.BOOK_FK_ID
        AND TABLE9.AUTHOR_FK_ID = TABLE8.AUTHOR_PK_ID
Group by
       TABLE6.TITLE,
       TABLE1.TITLE,
       TABLE8.LASTNAME,
       TABLE1.YEAR
```

Figure 4-12: SQL query for the XMLQL query of Figure 4-7

Thus, the output of the QT is a valid SQL query. The SQL query for the given XMLQL query of Figure 4-7 is shown in Figure 4-12. The XMLQL query in Figure 4-7 requested information about the titles of books and articles, which have been written by authors having the same lastname and have been published in the year 1995. As can be seen from Figure 4-12, the SQL query *select*s “Lastname” from the AUTHOR table, “Title” and “Year” from the ARTICLE table and “Title” from the BOOK table, as these are the information requested by the XMLQL query. Note that the “Year” can be selected from either the ARTICLE table or the BOOK table, as they have the same values. The conditions/filters of the XMLQL query is present in the SQL query in the first two lines of the *where* clause. There is a self join on the AUTHOR table, but the *where* clause maps one AUTHOR table with the BOOK table and the other AUTHOR table with the ARTICLE table. Thus only those tuples which have the last names of the authors of books and authors the same are selected. The result set is further constrained by selecting only tuples which have the value of the year as “1995.” Thus the result set obtained from the SQL query will contain the same information as the XML document obtained as a result of the XMLQL query.

One of the features of XMLQL is that it allows us to create a new set of elements instances to hold the queried data. This operation of XMLQL is called as *Restructuring*. This can be achieved by the template expression in the CONSTRUCT clause. Hence the regular path expression of the bound variable can be modified to create a new document from the result. To take care of this, whenever a variable is selected in the SQL query, it is selected with an alias using the *AS* option. This is seen in the SQL query of Figure 4-12. The alias in the *AS* option is mapped to the regular expression of the bound variable in the CONSTUCT clause. Thus when the relational result is converted to XML format, the tags created are based on the regular expression mappings of the aliases of the columns.

The correctness of the SQL query generated is verified in the next Chapter. While creating the SQL Query Creating Algorithm, there were other complexities encountered which have been resolved. Although no information is present regarding the datatypes in both SQL and XML, based on the relational operator in the condition associated with an bound element in the XMLQL query, if "<" , ">" , "<=" , and ">=" is present, then that bound variable has to be converted to number for the condition in the *where* clause of the

SQL query. Though XMLQL can bind a bound tag with just PCDATA or with sub-elements, in the SQL Query Creation, the bound element must be mapped to only columns. If the bound tag is a TABLE in the relational schema, select the table and all the columns, only if the table has not been selected. If a Table has been selected, then only those columns not already present in the *select* clause must be selected.

4.3 Maintenance Query Generator (MQG)

The data warehouse needs to be maintained with up-to-date information so that in future user queries may be satisfied with the cached data. In case the DM decides that the warehouse cannot satisfy the user query, the sources have to be queried. Hence the MQG creates a maintenance query to obtain the current content of the sources so as to load and update the warehouse.

We present an algorithm to generate maintenance queries based on information that indicates the data in the warehouse is not up-to-date. As explained in Chapter 2, we allow the user to decide a tolerance limit, and when the tolerance limit is reached, it is assumed that the data in the warehouse is stale. This tolerance limit is specified by keeping timestamps of all the elements in the global schema. The timestamps indicate when the element was last updated in the warehouse. The functionality of MQG is to generate a maintenance query and invoke the loader with the desired loading option on the resulting XML document. Also, it needs to update the timestamps of those elements which are obtained as a result of the maintenance query. The algorithm to generate maintenance queries is shown in Figure 4-13.

The input to the algorithm is the *Maintenance hashtable*, a data structure created at *built-time* (Figure 4-1). The hashtable consists of a set of elements for which maintenance queries must be generated and the path of these elements from the root in the global schema. These elements can be present at any level in the hierarchical structure of the global schema.

```
1   For each tag in the User Query
2        Check if it a queriable tag present in the Maintenance Hashtable
3        If so,
4          Store both the Tag and its path in the UserMaintenance Hashtable

5   For each element in the UserMaintenance Hashtable, a maintenance query
     must be generated
6        Select element from the UserMaintenance Hashtable and its path
7        Create a XMLQL query querying for the element by constructing the
          tag elements in both the WHERE and the CONSTRUCT clause
8        Use the Update TimeStamp Algorithm to update the TimeStamps of
          the tag variables queried
9        Add this XMLQL query to the set of Maintenance Queries generated
```

Figure 4-13: Pseudo code of the Maintenance Query Generation algorithm

Let us understand how the algorithm works. As before, the Line numbers assist us in understanding the algorithm. When the user query is parsed to identify those elements which are being queried, the element is checked to see whether it is a queriable tag present in the *Maintenance hashtable*. This is indicated in Line 2 of Figure 4-13. If it is a queriable tag, then both the tag and path of the tag from the root in the hierarchical structure are stored in a *UserMaintenance hashtable* (Line 4). This is repeated for all the bound elements in the user query, as indicated by Line 1.

A maintenance query must be generated for each of the element in the *UserMaintenance hashtable* (Line 5). For each element in this hashtable, obtain the path from the hashtable (Line 6). An XMLQL query is generated with that element as the bound variable, as shown in Line 7. This is added to the set of maintenance queries generated (Line 9). The timestamps of those variables for which the data is requested is then updated (Line 8).

After the result of the user query is obtained from the Mediator, and sent back to the Querying and Browsing Interface, the maintenance queries are sent to the Mediator. The Loader module of WHM is then invoked with the resulting document for loading the warehouse.

The timestamps of those elements that are present in the maintenance queries must be updated to indicate the freshness of data in the warehouse. The pseudocode of the algorithm is shown in Figure 4-14.

```
1   The parameter sent by the Maintenance Query Algorithm is a table
2   For all columns in the Table,
3       Update the timestamps of the tableColumn element in the
         TimeStamp Info Hashtable
4   For all tables associated with this table
5       Call the algorithm recursively with the associated table name
       as the parameter
```

Figure 4-14: Pseudo code of the Update Time Stamp algorithm

The element queried in a maintenance query is a table in the relational schema (Line 1). Hence for all the columns of the table (Line 2), we identify the tableColumn element in the *TimeStamp Info hashtable*, and update the timestamps with the current time. This is explained in Line 3. Since this table can have other tables associated with it in the relational mapping of the global schema, for all the tables associated with this table (Line 4), this algorithm is called recursively to update the timestamps of the columns of the associated tables, as described in Line 5 of the algorithm.

Updating the contents of the warehouse provides persistence, faster processing of frequently asked queries without having to go to the sources, and automatic maintenance of data in the warehouse.

CHAPTER 5
QUALITATIVE ANALYSIS

In this chapter, we shall look into the correctness of the queries generated by the XMLQL-2-SQL Query Translator (QT) and Maintenance Query Generator. The tests for the QT are based on standard query operations of any query language i.e., selection, extraction etc., These operations are explained in detail later in this chapter. The correctness of the query translation can be checked by comparing the output of the XMLQL query on the XML document and the output of the SQL query on data that was loaded into the warehouse. Semistructured query languages such as XMLQL have a lot more flexibility than SQL. We shall first give the setup on which the test queries were run. In the next section, we shall explain the characteristics required by an XMLQL query language and how each of this characteristics is tested for correctness in the query transformation. Finally, we shall perform tests to verify the correctness of the maintenance queries generated by the Maintenance Query Generator.

5.1 Experimental Setup

The IWiz testbed resides in the Windows NT environment. The experiments are carried out using a Pentium II 233 MHz processor with 128 MB of main memory. The components of this research were implemented using Java (SDK 1.3) from Sun Microsystems. Some of the other tools we use are the XML Parser from Oracle version 2.0.2.9, Oracle 8*i* as the warehouse, and AT&T Bell Labs' implementation of XMLQL processor version 0.9. All components communicate with each other using Java RMI.

IWiz has been implemented using Java, allowing it to be implemented across different platforms.

5.2 Query Operations and Test Queries

To evaluate the performance of a system, several prerequisites are needed. However, at this point in the XML world, practically all the prerequisites like sets of suitable data, sets of benchmark programs, and performance characteristics observed by comparable systems during testing are non-existent**.** The only XML-processing programs that are benchmarked are several XML parsers [8]. There is little material for benchmarking of XML data management systems [32]. For the purpose of testing, we use several XML sources containing bibliographic data.

We shall now describe the testing of the query translations generated by the QT. The general technique of translating an XMLQL query into a SQL query is as follows: (a) first, the relations corresponding to the start of the root path expressions are identified and added to the *From* clause of the SQL query then (b) the path expressions are translated to joins among relations. We evaluate the performance of this approach to query XML data with a set of queries that satisfy all operations that need to be performed by an XMLQL query.

```
WHERE
  <Bib>
    <Book year=$y>
      <Publisher>"Addison-Wesley"</Publisher>
      <Title> $t </Title>
      <Author> $a </Author>
    </Book>
  </Bib> IN “books.xml”,
  $y > 1991
CONSTRUCT $a
```

Figure 5-1: XMLQL Query Operation – *Selection* and *Extraction*

The different operations that have to be supported by an XML query language are selection, extraction, reduction, restructuring and combination [25]. These terms are explained below with queries to indicate the operations performed.

5.2.1 Selection and Extraction

Selection is the process of choosing a document element based on content, structure or attributes. *Extraction* is the process of pulling out particular elements of a document.

```
Select
  TABLE1.LASTNAME As sql2,
  TABLE1.FIRSTNAME As sql1,
  TABLE1.ADDRESS As sql3
From
  BOOK_AUTHOR TABLE5,
  BIB TABLE2,
  BIB_BOOK TABLE4,
  BOOK TABLE3,
  AUTHOR TABLE1
Where
  to_number(TABLE3.YEAR,'99999999999') > 1991
  AND TABLE3.PUBLISHER LIKE '%Addison-Wesley%'
  AND TABLE2.BIB_PK_ID = TABLE4.BIB_FK_ID
  AND TABLE4.BOOK_FK_ID = TABLE3.BOOK_PK_ID
  AND TABLE3.BOOK_PK_ID = TABLE5.BOOK_FK_ID
  AND TABLE5.AUTHOR_FK_ID = TABLE1.AUTHOR_PK_ID
Group by
  TABLE1.LASTNAME,
  TABLE1.FIRSTNAME,
  TABLE1.ADDRESS
```

Figure 5-2: SQL *Selection* and *Extraction* corresponding to Figure 5-1

Selection in XMLQL is done using patterns and conditions. Figure 5-1 shows a sample XMLQL query that selects all books published by Addison-Wesley after 1991. *Extraction* is done with the bound variables. The query binds the variables $t, $a and $y. <Publisher> is an element that is being selected and it must have the content “Addison-Wesley.” Also, the bound variable $y has the constraint that it should be greater than 1991. The query extracts the contents of author ($a).

The SQL query created by the QT, as a result of the XMLQL query of Figure 5-1, is shown in Figure 5-2. As can be seen from the figure, the *selection* operation of an XMLQL query is the *from* and *where* clause of a SQL query. The *extraction* operation of an XMLQL query is the *select* clause of a SQL query.

The *select* clause selects information from the table AUTHOR, as required by the XMLQL query and the *where* clause has the constraint that the content of the column PUBLISHER in the table BOOK must be "Addison-Wesley" and the content of the column YEAR in the table BOOK must be greater than 1991. Although the variable title ($t) is bound in the XMLQL query, it is not selected in the SQL query as it is not required. Thus the SQL query generated matches requests for the same information as that requested by the XMLQL query.

```
WHERE
  <Bib>
    <Book year=$y>
      <Publisher>"Addision-Wesley"</Publisher>
      <Title> $t </Title>
      <Author> $a </Author>
    </Book>
  </Bib> IN “books.xml”,
  $y > 1991
CONSTRUCT
  <result>
    <author> $a </author>
    <title> $t </title>
  </result>
```

Figure 5-3: XMLQL Query Operation – *Reduction* and *Restructuring*

5.2.2 Reduction and Restructuring

Reduction is the process of removing selected sub-elements of an element. *Restructuring* is constructing a new set of element instances to hold queried data. Figure 5-3 shows the same query as Figure 5-1, but with additional structure in the CONSTRUCT clause to output the author and title of a book. *Reduction* is achieved by controlling what elements are returned by the CONSTRUCT clause. In the query, only information about the author and the title are returned. *Restructuring* is controlled by the template expression of the CONSTRUCT clause. The querying of author and title of books in bib is restructured with the root element being "result"

```
Select
  TABLE1.LASTNAME As sql2,
  TABLE1.FIRSTNAME As sql1,
  TABLE1.ADDRESS As sql3,
  TABLE3.TITLE As sql4
From
  BOOK_AUTHOR TABLE5,
  BIB TABLE2,
  BIB_BOOK TABLE4,
  BOOK TABLE3,
  AUTHOR TABLE1
Where
  to_number(TABLE3.YEAR,'9999999') > 1991
  AND TABLE3.PUBLISHER LIKE '%Addison-Wesley%'
  AND TABLE2.BIB_PK_ID = TABLE4.BIB_FK_ID
  AND TABLE4.BOOK_FK_ID = TABLE3.BOOK_PK_ID
  AND TABLE3.BOOK_PK_ID = TABLE5.BOOK_FK_ID
  AND TABLE5.AUTHOR_FK_ID = TABLE1.AUTHOR_PK_ID
Group by
  TABLE1.LASTNAME,
  TABLE1.FIRSTNAME,
  TABLE1.ADDRESS,
  TABLE3.TITLE
```

Figure 5-4: SQL *Reduction* and *Restructuring* corresponding to Figure 5-3

In the SQL query created by QT and shown in Figure 5-4, *reduction* is performed by selecting those variables that are present in the CONSTRUCT clause. *Restructuring* is taken care by giving aliases to the elements in the *Select* clause of the SQL query and then creating the XML document from the result set by using these aliases to map the regular expression of the path in the CONSTRUCT clause.

For the given XMLQL query, information about year ($y), publisher ("Addison-Wesley"), title ($t) and author ($a) are queried, but only information about the title and author are given back to the user. This *reduction* is taken care in the SQL query by *select*ing only LASTNAME, FIRSTNAME and ADDRESS from the AUTHOR table and TITLE from the BOOK table. Information about either the year or the publisher is not selected. Also, when these columns are being selected in the *Select* clause, they are given aliases. These aliases are then used to map the element instances as present in the CONSTRUCT clause, so that the resulting XML document from the relational result set is restructured to the user specifications.

This query also illustrates the preservation of association: authors and titles are grouped as they appear in the input document.

```
WHERE
  <Ontology>
    <Bib>
      <Book>$book</Book>
    </Bib>
  </Ontology> IN “books.xml”,
  <Title> $title </Title> IN $book,
  <Publisher>"Addison-Wesley"</Publisher> IN $book
CONSTRUCT
  <book>
    <title>$title</title>
    <authors>
      { WHERE <author>$author</author> IN $book
        CONSTRUCT $author }
    </authors>
  </book>
```

Figure 5-5: XMLQL Query Operation – *Complex Restructuring*

5.2.3 Complex Restructuring

In the previous query, we see that association property is preserved by grouping each author with its corresponding title. But if a document has multiple instances of author the resulting document will contain each instance of the author coupled with the title. To eliminate this, the query of Figure 5-5 is written which is restructured to group results by book title. It is written using a nested query.

```
Select
  TABLE7.LASTNAME As sql3,
  TABLE7.FIRSTNAME As sql2,
  TABLE7.ADDRESS As sql4,
  TABLE2.TITLE As sql1
From
  BOOK_AUTHOR TABLE8,
  BIB_BOOK TABLE6,
  ONTOLOGY_BIB TABLE5,
  BOOK TABLE2,
  ONTOLOGY TABLE3,
  AUTHOR TABLE7,
  BIB TABLE4
Where
  TABLE2.PUBLISHER LIKE '%Addison-Wesley%'
  AND TABLE3.ONTOLOGY_PK_ID = TABLE5.ONTOLOGY_FK_ID
  AND TABLE5.BIB_FK_ID = TABLE4.BIB_PK_ID
  AND TABLE4.BIB_PK_ID = TABLE6.BIB_FK_ID
  AND TABLE6.BOOK_FK_ID = TABLE2.BOOK_PK_ID
  AND TABLE2.BOOK_PK_ID = TABLE8.BOOK_FK_ID
  AND TABLE8.AUTHOR_FK_ID = TABLE7.AUTHOR_PK_ID
Group by
  TABLE7.LASTNAME,
  TABLE7.FIRSTNAME,
  TABLE7.ADDRESS,
  TABLE2.TITLE
```

Figure 5-6: SQL *Complex Restructuring* corresponding to Figure 5-5

In the XMLQL query, the first WHERE clause bins the variable $book to the contents of `<book>`…`</book>`. For each such binding, one `<book>` and one `<title>` are emitted. Then the inner where clause is evaluated, which, in turn, produces one or several authors. The QT takes care of this condition while generating the SQL query, by grouping the results first by the inner most nested query, then in the decreasing order of the nested query. The SQL query is shown in Figure 5-6, containing the *group by* clause, which takes care of restructuring. Thus, grouping them in this form would make the resulting XML document from the result set to be nested in the order that the query has specified.

```
WHERE
  <Ontology>
    <Bib>
      <Book>
        <Title> $title_book </Title>
        <Author>
          <Lastname>
            <PCDATA> $author_book </PCDATA>
          </Lastname>
        </Author>
        <Year> $year </Year>
      </Book>
      <Article>
        <Title> $title_article </Title>
        <Author>
          <Lastname>
            <PCDATA> $author_article </PCDATA>
          </Lastname>
        </Author>
        <Year> $year </Year>
      </Article>
    </Bib>
  </Ontology> IN “books_article.xml”,
  $year = 1995,
  $author_book = $author_article
CONSTRUCT
  <authors_of_book_or_article year=$year>
      $author_book
    <book_title>$title_book </book_title>
    <article_title>$title_article</article_title>
  </authors_of_book_or_article>
```

Figure 5-7: XMLQL Query Operation – *Combination*

5.2.4 Combination

Combination is the operation of merging two or more elements into one. Figure 5-7 indicates an XMLQL query with one such operation. Here, we have two different elements, a `<book>` element and an `<article>` element. The query requests for information about books and articles that were written in the same year 1995, and by authors who had the same lastnames. Specifically, the information requested is the titles of the book and article and the author’s lastnames. This XMLQL query reads data from both elements and computes a “join.” The join value is the common year $year, and the bound variables $author_book and $author_article which must be the same. The query

```
Select
  TABLE10.LASTNAME As sql4,
  TABLE1.TITLE As sql1,
  TABLE7.TITLE As sql3,
  TABLE7.YEAR As sql2
From
  BIB TABLE3,
  ONTOLOGY TABLE2,
  BIB_ARTICLE TABLE8,
  ARTICLE TABLE7,
  AUTHOR TABLE6,
  BOOK TABLE1,
  ONTOLOGY_BIB TABLE4,
  BIB_BOOK TABLE5,
  AUTHOR TABLE10,
  ARTICLE_AUTHOR TABLE9,
  BOOK_AUTHOR TABLE11
Where
  TABLE10.LASTNAME LIKE TABLE6.LASTNAME
  AND TABLE7.YEAR LIKE '%1995%'
  AND TABLE1.YEAR LIKE '%1995%'
  AND TABLE2.ONTOLOGY_PK_ID = TABLE4.ONTOLOGY_FK_ID
  AND TABLE4.BIB_FK_ID = TABLE3.BIB_PK_ID
  AND TABLE3.BIB_PK_ID = TABLE5.BIB_FK_ID
  AND TABLE5.BOOK_FK_ID = TABLE1.BOOK_PK_ID
  AND TABLE3.BIB_PK_ID = TABLE8.BIB_FK_ID
  AND TABLE8.ARTICLE_FK_ID = TABLE7.ARTICLE_PK_ID
  AND TABLE7.ARTICLE_PK_ID = TABLE9.ARTICLE_FK_ID
  AND TABLE9.AUTHOR_FK_ID = TABLE6.AUTHOR_PK_ID
  AND TABLE1.BOOK_PK_ID = TABLE11.BOOK_FK_ID
  AND TABLE11.AUTHOR_FK_ID = TABLE10.AUTHOR_PK_ID
Group by
  TABLE1.TITLE,
  TABLE7.TITLE,
  TABLE10.LASTNAME,
  TABLE7.YEAR
```

Figure 5-8: SQL *Combination* corresponding to Figure 5-7

tries every match in the first data element against every match in the second data element, and checks if they have the same year and the lastnames of authors match: if yes, the requested data is output. The QT translates this query into a query as shown in Figure 5-8.

In the relational schema, as all elements are mapped to tables or columns, a *combination* of two elements refers to joins on two tables. We use the *Relational mapping tables* to map the different tables as explained in Chapter 4.

5.3 Analysis of the Results

The above examples indicate the operations that must be done by an XML query language. A sample XML document was taken containing information about books and articles and loaded into the data warehouse by WHM. The XMLQL queries were given as input to the XMLQL-2-SQL Query Translator, which generated the corresponding SQL queries. The SQL query generated was executed on the data in the warehouse, while the XMLQL queries were executed on the XML document using AT&T's implementation of XMLQL processor. The resulting XML document was then checked against the relational set obtained from the SQL query. The correctness of the SQL query was verified by the fact that the results of both the XMLQL query and the SQL query were the same.

The minimal representational features in the DTD limit the datatypes of all fields in the relational schema to be of type *varchar*. This causes a constraint for conditions related with data of *numeric* type, as seen in Figure 5-2 and Figure 5-4 (*to_number(TABLE3.YEAR,'99999999999') > 1991*).

The *varchar* has to be converted to a *number* datatype using the *to_number* function. But the problem faced is to identify how many digits would be present in the number. A set number of digits must be hardcoded when we use a DTD. Using a XML schema instead of a DTD when creating the relational schema would be useful in that we can know the datatype of the element being stored and no assumptions need to be made.

Another problem faced during XMLQL to SQL translation is that, while loading the data into the warehouse, it may be loaded with spaces either in front or at the end of the actual data. Hence all conditions must take care of these spaces. In the SQL query generated, the “%” sign before and after the condition makes sure that the conditions are checked correctly and the output generated as required.

5.4 Analysis of Maintenance Queries

When the Decision Module obtains the user query and decides that the queries cannot be satisfied by the warehouse, it sends the query to the sources. To update the contents of the warehouse, it calls the Maintenance Query Generator to generate maintenance queries. The result obtained due to the maintenance queries is then loaded into the warehouse. When a query requesting similar information is given to WHM, because of the freshness of data in the warehouse, DM decides that the warehouse can satisfy the query and queries the warehouse.

```
WHERE
  <Bib>
    <Book year=$y>
      <Publisher>"Addison-Wesley"</Publisher>
      <Title> $t </Title>
    </Book>
  </Bib> IN “books.xml”,
  $y > 1991
CONSTRUCT $t
```

Figure 5-9: An XMLQL query requesting information about book

The decision module was tested by sending the same query from the Querying and Browsing Interface to WHM. The first time, since there was no data, the sources were queried and the result returned to the user. The maintenance query was then sent to the sources and the result loaded into the warehouse. When the same query was sent again, since the warehouse was updated, DM sent the XMLQL query to the XMLQL-2-SQL Query Translator to translate the query into SQL and query the warehouse. Repetitions of such tests verified the correct working of DM.

```
function query() {
  CONSTRUCT
  <Ontology> {
      WHERE
         <Ontology>
           <Bib>
             <Book> $Book </Book>
           </Bib>
         </Ontology> IN Mediator
      CONSTRUCT
       <Bib>
         <Book> $Book </Book>
       </Bib>
    }
  </Ontology>
}
```

Figure 5-10: Maintenance Query for Figure 5-9

An XMLQL query requesting information about books is shown in Figure 5-9. When this query cannot be satisfied by the warehouse, a maintenance query has to be generated. DM identifies what is the data requested by the user and using the *Maintenance hashtable* (which is obtained at *built-time* as explained in Chapter 4), generates a maintenance query, shown in Figure 5-10. As can be seen, when information in any of the attributes (e.g., publisher, title, year) of a queriable tag (e.g., book) is not present or is stale in the warehouse, a maintenance query is generated requesting *all* information about book.

When the data in the warehouse was stale, another query requesting information about authors of book was given to DM by the Querying and Browsing Interface. The nested XMLQL query is shown in Figure 5-11, and the MQG generated a maintenance query which was the same as shown in Figure 5-10. This indicated the correctness of the MQG in that even if any attribute of a queriable tag was considered absent or the data stale in the warehouse, then all information about the queriable tag was requested. The same type of maintenance queries was generated irrespective of whether it was a simple or nested query with conditions and filters.

```
WHERE
  <Ontology>
    <Bib>
      <Book>$book</Book>
    </Bib>
  </Ontology> IN “books.xml”
CONSTRUCT
  <book>
    <authors>
      { WHERE <author>$author</author> IN $book
        CONSTRUCT $author }
    </authors>
  </book>
```

Figure 5-11: Nested XMLQL query requesting information about authors of book

Since all types of operations are possible in a single XML query, our approach to create SQL queries and maintenance queries must take care of these conditions and operations. Based on the experimental results, our integration system supports all basic operations on simple and nested queries with filters and conditions. However, operations like recursive definitions and regular path expressions cannot be satisfied by the current implementation. But, the robustness and efficiency obtained with increased volumes of data weigh in selecting relational databases for storing and querying frequently accessed simple data.

CHAPTER 6
CONCLUSIONS

Research on semistructured data is aiming to extend database management techniques to include data with irregular, unknown structure. Integration of heterogeneous data sources is a very complex and challenging task and methodologies for providing efficient and effective integration are in great demand. The intent of this thesis is to provide a solution to the problem of integrating XML with RDBMS.

In this thesis, we provide algorithms and implementation details of the XMLQL-2-SQL Query Translator (QT), the Decision Module (DM), and the Maintenance Query Generator (MQG), which are part of the Warehouse Manager (WHM) component of the IWiz prototype. The internal data model and query language used in IWiz is XML and XMLQL. The data warehouse is implemented using Oracle 8*i*.

6.1 Contributions

Through this research, we have identified a method to analyze the user query to verify whether the data warehouse can satisfy the query requirements. If the warehouse can satisfy the user query, translate the XML query on the XML view into SQL query on the data stored in the relational database. If not, send the user query to the sources through the mediator and generate maintenance queries based on information that the data in the warehouse is not up-to-date.

The fundamental differences between structure of an XMLQL query and a SQL query has been resolved. XMLQL can have different types of filters and conditions such as implicit and explicit filters and conditions, which must be translated to *where* conditions of a SQL query. Also, these filters and conditions can be present with/without quotes which must be resolved.

Another issue is that XMLQL can bind a variable to an element which has PCDATA as its child or a set of sub-elements. So, the mapping of the element to a table or column in the relational schema must be discovered. If the element was mapped to a table, then all the columns of the table must be selected.

In XML, all elements are considered to have a datatype of string. Hence joins and filters with relational operators like <, >, <=, and >= pose a problem when translated into a SQL query since the datatype associated with these operators are numbers. This problem has been identified and taken care of in our implementation of QT.

Operations like *restructuring*, which is taken care of in the XMLQL query by the CONSTRUCT clause should also be satisfied. We have implemented the method of aliasing the variables of the *select* clause of the SQL query with the mapping of the element from the root of the global schema. The XML document is created from the relational result obtained, by using the mappings to restructure the document.

The principal contributions of this thesis are as follows. We have designed and implemented an algorithm to automate the process of deciding whether the information requested by the user is stored in the data warehouse, or if it has to be fetched from the underlying sources in real-time. The principal focus of this research was to design an algorithm to translate the user queries from their XMLQL representation into SQL and process them in relational systems without manual intervention. In case the sources have to be queried, we have developed an algorithm to create a maintenance query to obtain the current content of the sources so as to load and update the contents of the warehouse. The updation of the contents of the warehouse provides persistence, faster processing of frequently asked queries, and automatic maintenance of data contents of the warehouse.

6.2 Future Work

The research done so far on semistructured data may offer some solutions to database problems posed by XML. But XML research on semistructured data has not yet addressed other problems like type inference, distributed evaluation or a proper storage mechanism.

Our approach uses DTDs as the mechanism for data definition. Hence the relational mapping is based on the hierarchical structure of the DTD. The richer data definition of XML Schema makes it superior over DTD and has been advanced to proposed recommendation status. Hence in future, our prototype may shift to using XML Schema as the global schema.

The current implementation of QT implements all basic features of an XMLQL query like *selection*, *extraction*, *flattening*, *preserving structure*, *restructuring*, *complex restructuring by nesting*, *changing structure by explicit grouping*. While most of the queries can be written with these functionalities, there are other features of XMLQL which are not currently handled. Primary among these, which must be taken care of would be *recursive* definitions, *tag variables* (the bound variable can be from a set of different elements), *sorting*. We assume that the next implementation of the QT will incorporate these features. Other features include *external functions* and *aggregation*.

Using relational storage for XML has received considerable attention. The performance of the relational approach depends on the effectiveness of the system's query optimizer . Semantically equivalent queries expresses as different SQL queries can cause relational databases to choose different execution plans which may degrade the performance. Our approach generates “correct plans,” though not necessarily optimal. Since the quality of query optimizers in commercial systems are high and with a fixed schema, and queries on XML documents exhibit predictable access patterns, we can implement a method to indicate what mapping strategy can be used for a given query.

APPENDIX

Ontology.DTD

```
<!ELEMENT Ontology (Bib)*>
<!ELEMENT Bib (Article | Book | Booklet | InBook | InCollection | InProceedings |
        Manual | MastersThesis | PhdThesis | Misc | Proceedings | TechReport |
        UnPublished )*>
<!ELEMENT Article ( Author+, Editor*, Title, Year, Month?, Pages?, Note?, Journal)>
<!ELEMENT Book ((Author+ | Editor+), Title, Publisher, Year, Month?, (Volume |
        Number1)?, Series?, Address?, Edition?, ISBN, Cost?, Note?)>
<!ELEMENT Booklet (Author*, Title, HowPublished?, Year?, Month?, Address?,
        Note?)>
<!ELEMENT InBook (Author+, Title, Year, Month?, Publisher,(Pages | Chapter?), Book
        )>
<!ELEMENT InCollection (Author+, Title, Year, Month?, (Pages | Chapter?), Type?,
        Note?, Collection)>
<!ELEMENT InProceedings (Author+, Title, Year, Month?, (Pages | Chapter?), Note?,
        Proceedings)>
<!ELEMENT Manual (Title, Author*, Year?, Month?, Edition?, Address?,
        Organization?, Note?)>
<!ELEMENT MastersThesis (Author, Title, School, Address?, Year, Month?, Type?,
        Note?)>
<!ELEMENT Misc (Author?, Title?, HowPublished?, Year?, Month?, Note?)>
<!ELEMENT PhdThesis (Author, Title, School, Address?, Year, Month?, Type?,
        Note?)>
<!ELEMENT TechReport (Author+, Title, Institution, Year, Month?, Type?, Number1?,
        Address?, Note?)>
<!ELEMENT UnPublished (Author+, Title, Year?, Month?, Note)>
<!ELEMENT Address (#PCDATA)>
<!ELEMENT Author (Firstname?, Lastname, Address?)>
<!ELEMENT Chapter (#PCDATA) >
<!ELEMENT Collection ((Author+ | Editor+), Title, Publisher?)>
<!ELEMENT Cost (#PCDATA)>
<!ELEMENT Editor (#PCDATA)>
<!ELEMENT Edition (#PCDATA)>
<!ELEMENT Firstname (#PCDATA)>
<!ELEMENT ISBN (#PCDATA)>
<!ELEMENT HowPublished (#PCDATA) >
<!ELEMENT Institution (#PCDATA) >
<!ELEMENT Journal (Title, Year?, Month?, Volume?, Number1? )>
```

```
<!ELEMENT Lastname (#PCDATA)>
<!ELEMENT Month (#PCDATA) >
<!ELEMENT Note (#PCDATA) >
<!ELEMENT Number1 (#PCDATA) >
<!ELEMENT Organization (#PCDATA) >
<!ELEMENT Pages (#PCDATA) >
<!ELEMENT Proceedings (Title, Editor*, Year?, Month?, (Volume | Number1)?,
        Series?, Address?, Organization?, Publisher?, Note?)>
<!ELEMENT Publisher (#PCDATA) >
<!ELEMENT School (#PCDATA) >
<!ELEMENT Series (#PCDATA)>
<!ELEMENT Title (#PCDATA) >
<!ELEMENT Type (#PCDATA) >
<!ELEMENT Volume (#PCDATA) >
<!ELEMENT Year (#PCDATA) >
```

<u>Maintenance hashtable</u>

Book	Ontology.Bib.Book
InProceedings	Ontology.Bib.InProceedings
Article	Ontology.Bib.Article
Journal	Ontology.Bib.Article.Journal
UnPublished	Ontology.Bib.UnPublished
TechReport	Ontology.Bib.TechReport
InCollection	Ontology.Bib.InCollection
Misc	Ontology.Bib.Misc
MastersThesis	Ontology.Bib.MastersThesis
InBook	Ontology.Bib.InBook
Manual	Ontology.Bib.Manual

<u>Tag Info hashtable</u>

InProceedings:Pages	INLINED-FIELD
Bib:InCollection	CONCEPT
Article:Journal	CONCEPT
PhdThesis:Year	INLINED-FIELD
Proceedings:Editor	CONCEPT
Collection:Publisher	INLINED-FIELD
Manual:Organization	INLINED-FIELD
UnPublished:Author	CONCEPT
PhdThesis:Title	INLINED-FIELD
InProceedings:Year	INLINED-FIELD
Bib:Book	CONCEPT
Bib:Manual	CONCEPT
Proceedings:Series	INLINED-FIELD

InCollection:Type	INLINED-FIELD
Article:Note	INLINED-FIELD
Manual:Author	CONCEPT
Misc:HowPublished	INLINED-FIELD
InCollection:Author	CONCEPT
Book:Publisher	INLINED-FIELD
MastersThesis:Note	INLINED-FIELD
InBook:Author	CONCEPT
Collection:Author	CONCEPT
Proceedings:Year	INLINED-FIELD
PhdThesis:Author	CONCEPT
InBook:Publisher	INLINED-FIELD
PhdThesis:Address	INLINED-FIELD
Book:Address	INLINED-FIELD
UnPublished:Note	INLINED-FIELD
MastersThesis:Month	INLINED-FIELD
Book:Edition	INLINED-FIELD
InBook:Month	INLINED-FIELD
Misc:Title	INLINED-FIELD
Booklet:Note	INLINED-FIELD
InCollection:Year	INLINED-FIELD
Misc:Note	INLINED-FIELD
Book:Month	INLINED-FIELD
Manual:Year	INLINED-FIELD
MastersThesis:Address	INLINED-FIELD
Manual:Month	INLINED-FIELD
Collection:Editor	CONCEPT
InProceedings:Month	INLINED-FIELD
Proceedings:Month	INLINED-FIELD
Collection:Title	INLINED-FIELD
InCollection:Title	INLINED-FIELD
InBook:Book	CONCEPT
InCollection:Pages	INLINED-FIELD
InCollection:Collection	CONCEPT
PhdThesis:Month	INLINED-FIELD
PhdThesis:Note	INLINED-FIELD
Proceedings:Number1	INLINED-FIELD
Bib:InProceedings	CONCEPT
Proceedings:Publisher	INLINED-FIELD
InBook:Chapter	INLINED-FIELD
InProceedings:Note	INLINED-FIELD
TechReport:Type	INLINED-FIELD
Bib:Article	CONCEPT
Ontology:Bib	CONCEPT
InProceedings:Proceedings	CONCEPT
Bib:Proceedings	CONCEPT

UnPublished:Title	INLINED-FIELD
Author:Lastname	INLINED-FIELD
Article:Title	INLINED-FIELD
Journal:Title	INLINED-FIELD
Article:Pages	INLINED-FIELD
InBook:Year	INLINED-FIELD
Booklet:HowPublished	INLINED-FIELD
Booklet:Author	CONCEPT
Proceedings:Note	INLINED-FIELD
MastersThesis:School	INLINED-FIELD
Book:Year	INLINED-FIELD
Bib:TechReport	CONCEPT
Editor:#text	INLINED-FIELD
TechReport:Year	INLINED-FIELD
Misc:Month	INLINED-FIELD
Journal:Year	INLINED-FIELD
Bib:Booklet	CONCEPT
Manual:Edition	INLINED-FIELD
InCollection:Note	INLINED-FIELD
TechReport:Title	INLINED-FIELD
Booklet:Title	INLINED-FIELD
Author:Address	INLINED-FIELD
TechReport:Author	CONCEPT
MastersThesis:Type	INLINED-FIELD
Journal:Number1	INLINED-FIELD
Manual:Note	INLINED-FIELD
Book:Author	CONCEPT
TechReport:Institution	INLINED-FIELD
Bib:UnPublished	CONCEPT
Bib:MastersThesis	CONCEPT
Manual:Address	INLINED-FIELD
InCollection:Month	INLINED-FIELD
InProceedings:Author	CONCEPT
Book:Cost	INLINED-FIELD
Proceedings:Address	INLINED-FIELD
Article:Year	INLINED-FIELD
TechReport:Number1	INLINED-FIELD
Booklet:Address	INLINED-FIELD
MastersThesis:Year	INLINED-FIELD
Author:Firstname	INLINED-FIELD
Book:Volume	INLINED-FIELD
Bib:InBook	CONCEPT
InCollection:Chapter	INLINED-FIELD
UnPublished:Year	INLINED-FIELD
InProceedings:Chapter	INLINED-FIELD
MastersThesis:Author	CONCEPT

Proceedings:Organization	INLINED-FIELD
Article:Author	CONCEPT
_:Ontology	CONCEPT
UnPublished:Month	INLINED-FIELD
Book:Editor	CONCEPT
Booklet:Year	INLINED-FIELD
Article:Month	INLINED-FIELD
MastersThesis:Title	INLINED-FIELD
Journal:Month	INLINED-FIELD
Bib:PhdThesis	CONCEPT
InBook:Title	INLINED-FIELD
Misc:Year	INLINED-FIELD
PhdThesis:School	INLINED-FIELD
InBook:Pages	INLINED-FIELD
Journal:Volume	INLINED-FIELD
PhdThesis:Type	INLINED-FIELD
Book:ISBN	INLINED-FIELD
Book:Note	INLINED-FIELD
Misc:Author	CONCEPT
Bib:Misc	CONCEPT
Proceedings:Volume	INLINED-FIELD
Book:Title	INLINED-FIELD
Book:Series	INLINED-FIELD
TechReport:Note	INLINED-FIELD
TechReport:Address	INLINED-FIELD
Manual:Title	INLINED-FIELD
TechReport:Month	INLINED-FIELD
Book:Number1	INLINED-FIELD
Booklet:Month	INLINED-FIELD
InProceedings:Title	INLINED-FIELD
Proceedings:Title	INLINED-FIELD

Table Info hashtable

InProceedings	[INPROCEEDINGS_PK_ID, TITLE, YEAR, MONTH, PAGES, CHAPTER, NOTE]
Journal	[JOURNAL_PK_ID, TITLE, YEAR, MONTH, VOLUME, NUMBER1]
Bib	[BIB_PK_ID]
Proceedings	[PROCEEDINGS_PK_ID, TITLE, YEAR, MONTH, VOLUME, NUMBER1, SERIES, ADDRESS, ORGANIZATION, PUBLISHER, NOTE]
MastersThesis	[MASTERSTHESIS_PK_ID, TITLE, SCHOOL, ADDRESS, YEAR, MONTH, TYPE, NOTE]

PhdThesis	[PHDTHESIS_PK_ID, TITLE, SCHOOL, ADDRESS, YEAR, MONTH, TYPE, NOTE]
InCollection	[INCOLLECTION_PK_ID, TITLE, YEAR, MONTH, PAGES, CHAPTER, TYPE, NOTE]
TechReport	[TECHREPORT_PK_ID, TITLE, INSTITUTION, YEAR, MONTH, TYPE, NUMBER1, ADDRESS, NOTE]
InBook	[INBOOK_PK_ID, TITLE, YEAR, MONTH, PUBLISHER, PAGES, CHAPTER]
UnPublished	[UNPUBLISHED_PK_ID, TITLE, YEAR, MONTH, NOTE]
Ontology	[ONTOLOGY_PK_ID]
Misc	[MISC_PK_ID, TITLE, HOWPUBLISHED, YEAR, MONTH, NOTE]
Author	[AUTHOR_PK_ID, FIRSTNAME, LASTNAME, ADDRESS]
Editor	[EDITOR_PK_ID, EDITOR]
Book	[BOOK_PK_ID, TITLE, PUBLISHER, YEAR, MONTH, VOLUME, NUMBER1, SERIES, ADDRESS, EDITION, ISBN, COST, NOTE]
Article	[ARTICLE_PK_ID, TITLE, YEAR, MONTH, PAGES, NOTE]
Manual	[MANUAL_PK_ID, TITLE, YEAR, MONTH, EDITION, ADDRESS, ORGANIZATION, NOTE]
Booklet	[BOOKLET_PK_ID, TITLE, HOWPUBLISHED, YEAR, MONTH, ADDRESS, NOTE]
Collection	[COLLECTION_PK_ID, TITLE, PUBLISHER]

Associated Table Info hashtable

InCollection	[Author, Collection]
TechReport	[Author]
Article	[Author, Journal]
Ontology	[Bib]
Collection	[Author, Editor]
Misc	[Author]
Booklet	[Author]
Manual	[Author]
InBook	[Author, Book]
Bib	[Article, Book, Booklet, InBook, InCollection, InProceedings, Manual, MastersThesis, Misc, PhdThesis, Proceedings, TechReport, UnPublished]
InProceedings	[Author, Proceedings]
Book	[Author, Editor]
UnPublished	[Author]
Proceedings	[Editor]
MastersThesis	[Author]
PhdThesis	[Author]

Time Stamp Info hashtable

ARTICLE:MONTH	2001-03-25 16:17:25.0
PROCEEDINGS:SERIES	2001-03-25 16:17:25.0
ARTICLE:AUTHOR	2001-03-25 16:17:25.0
AUTHOR:LASTNAME	2001-03-25 16:17:25.0
BOOKLET:MONTH	2001-03-25 16:17:25.0
BOOK:NOTE	2001-03-25 16:17:25.0
ONTOLOGY:BIB	2001-03-25 16:17:25.0
BIB:ARTICLE	2001-03-25 16:17:25.0
_:ONTOLOGY	2001-03-25 16:17:25.0
BIB:BOOK	2001-03-25 16:17:25.0
BOOK:TITLE	2001-03-25 16:17:25.0
BOOK:MONTH	2001-03-25 16:17:25.0
BOOK:ISBN	2001-03-25 16:17:25.0
BOOK:YEAR	2001-03-25 16:17:25.0
AUTHOR:ADDRESS	2001-03-25 16:17:25.0
BOOK:PUBLISHER	2001-03-25 16:17:25.0

LIST OF REFERENCES

1. S. Abiteboul, D. Quass, J. McHugh, J. Widom, J. Wiener, "The Lorel Query Language for Semistructured Data," *International Journal of Digital Libraries*, vol.1, 1997.

2. C. Beeri, T. Milo, "Schemas for Integration and Translation of Structured and Semi-Structured Data," *Proceedings of the International Conference on Database Theory*, Jerusalem, Israel, 1999.

3. S. Bergamaschi, S. Castano, M. Vincini, "Semantic Integration of Semistructured and Structured Data Sources," *SIGMOD Record*, vol.28, 1999.

4. P. Buneman, S. Davidson, G. Hillebrand, D. Suciu, "A query language and optimization techniques for unstructured data," *Proceedings of ACMSIGMOD International Conference on Management of Data*, pp. 505-516, Montreal, Canada, 1996.

5. D. Chamberlin, J. Robie, and D. Florescu, "Quilt: An XML Query Language for Heterogeneous Data Sources, " *In ACM SIGMOD Workshop on The Web and Databases (WebDB'00)*, Dallas, Texas, pp. 53-62, May 2000.

6. N. Clayton, "DTDs: Walkthrough of a simple DTD," June 2001, available at http://www.nothing-going-on.demon.co.uk/SGML/DTDs/simple.html

7. S. Cluet, J. Simeon, "YATL: a functional and declarative language for XML," Draft manuscript, Mar 2000.

8. C. Cooper, "Benchmarking XML Parsers: A performance comparison of six stream-oriented XML parsers," June 2001, available at http://www.xml.com/pub/a/Benchmark/article.html .

9. E. Derksen, P. Fankhauser, E. Howland, G. Huck, I. Macherius, M. Murata, M. Resnick, H. Schöning, "XQL (XML Query Language)," Submission to the World Wide Web Consortium, June 2001, available at http://www.ibiblio.org/xql/xql-proposal.html.

10. A. Deutsch, M. Fernandez, D. Florescu, A. Levy, D. Maier, and D. Suciu. "Querying XML data, " *Data Engineering Bulletin*, Vol. 22, No 3 pp.10-18, 1999

11. A. Deutsch, M. Fernandez, D. Florescu, A. Levy, D. Suciu, "XML-QL: A Query Language for XML," *Proceedings of 8th International World Wide Web Conference (WWW8)*, 1999.

12. A. Deutsch, M. Fernandez, D. Suciu, "Storing semistructured data with STORED," *Proceedings of ACM SIGMOD International Conference on Management of Data* , Philadelphia, Pennsylvania, pp. 431-442, 1999.

13. M. Fernandez, D. Florescu, J. Kang, A. Levy, D. Suciu, "Catching the boat with strudel: Experiences with a web-site management system, " *In Proceedings. of the ACM SIGMOD Conference. on Management of Data*, pp. 414-425, 1998

14. M. Fernandez, J. Simeon, P. Walder, S. Cluet, A. Deutsch, D. Florescu, A. Levy, D. Maier, J. McHugh, J. Robie, D. Suciu, J. Widom, "XML Query Languages: Experiences and Exemplars", June 2001, available at http://www-db.research.bell-labs.com/user/simeon/xquery.html.

15. D. Florescu, D. Kossmann, "A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database, " INRIA Technical Report, INRIA, No. 3680, May, 1999

16. D. Florescu, D. Kossmann, "Storing and Querying XML Data Using an RDBMS," *Data Engineering Bulletin, Special Issue on XML*, Vol. 22, No 3, pp. 27-34, September 1999

17. D. Florescu, A. Levy, A. Mendelzon, "Database techniques for the World-Wide Web: A survey," *SIGMOD Record*, Vol. 27, No. 3 pp. 59-74, 1998.

18. H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, J. Widom, "Integrating and Accessing Heterogeneous Information Sources in TSIMMIS," presented at *AAAI Symposium on Information Gathering*, Stanford, California, 1995.

19. GMD: GDM-IPSI XQL Engine, June 2001, available at http://xml.darmstadt.gmd.de/xql/.

20. R. Goldman, J. McHugh, J. Widom, "From Semistructured Data to XML: Migrating the Lore data Model and Query Language," *Proceedings of the 2nd International Workshop on the Web and Databases*, 1999.

21. J. Hammer, "The Information Integration Wizard (IWiz) Project," Project Description TR99-019, University of Florida, Gainesville, Florida, 1999.

22. J. Hammer, H. Garcia-Molina, W. Labio, J. Widom, Y. Zhuge. "The Stanford Data Warehousing Project", *Data Engineering Bulletin, Special Issue on Materialized Views and Data Warehousing*, vol. 18, pp. 41-48, 1995.

23. IBM's alphaWorks, "XML Parser for Java," June 2001, http://alphaworks.ibm.com/tech/xml4j.

24. W. J. Labio, Y. Zhuge, J. L. Wiener, H. Gupta, H. Garcia-Molina, and J. Widom, "The WHIPS Prototype for Data Warehouse Creation and Maintenance," *SIGMOD Record (ACM Special Interest Group on Management of Data)*, vol. 26, pp. 557-559, 1997.

25. D. Maier, "Database desiderata for an XML Query Language," *In Proceedings of the Query Languages workshop QL'98*, Cambridge, Mass., Dec. 1998

26. I. Manolescu, D. Florescu, D. Kossmann, "Pushing XML queries inside relational databases," Tech. Report no. 4112, INRIA, 2001

27. I. Manolescu, D. Florescu, D. Kossmann, F. Xhumari, D. Olteanu, "Agora: Living with XML and Relational," pp. 623-626, *Software Demonstration at VLDB 2000*.

28. D. Martin, M. Birbeck, M. Kay, B. Loesgen, J. Pinnock, S. Livingstone, P. Stark, K. Williams, R. Anderson, S. Mohr, D. Baliles, B. Pear, N. Ozu, "Professional XML," Wrox Press, 2000.

29. J. McHugh, S. Abiteboul, R. Goldman, D. Quass, J. Widom, "Lore: A Database Management System for Semistructured Data", *SIGMOD Record*, vol. 26, pp. 54-66, 1997.

30. Oracle XML SQL Utility (XSU) for Java and XSQL Servlet, June 2001, available at http://technet.oracle.com/tech/xml.

31. D. Quass, J. Widom, "On-Line Warehouse View Maintenance for Batch Updates," *In Proceedings of the ACM SIGMOD Conference*, pp. 147-158, Tucson, Arizona, May, 1997

32. A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse. "The XML Benchmark Project," Technical Report INS-R0103, CWI, Amsterdam, The Netherlands, April 2001

33. J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, J. Naughton, "Relational Databases for Querying XML Documents: Limitations and Opportunities", *Proceedings of the 25th Conference on Very Large Databases*, Edinburg, Scotland, 1999.

34. Software AG: Tamino – The Information Server for Electronic Business, June 2001, available at http://www.softwareag.com/tamino/

35. D. Suciu, "An Overview of Semistructured Data," SIGACT News, 29(4), 1998.

36. F. Tian, D. DeWitt, J. Chen, and C. Zhang, "The Design and Performance Evaluation of Alternative XML Storage Strategies," Technical report, University of Wisconsin, 2000, Available at http://www.cs.wisc.edu/niagara/papers/vldb00XML.pdf

37. J. Widom, "Data Management for XML: Research Directions," *Data Engineering Bulletin, Special Issue on XML*, Vol. 22, No 3, September 1999.

38. J. Widom, "Research Problems in Data Warehousing," *Fourth International Conference on Information and Knowledge Management*, Baltimore, MD, 1995.

39. World Wide Web Consortium, "Document Object Model (DOM) Level 1 Specification," W3C Recommendation, June 2001, available at http://www.w3.org/TR/REC-DOM-Level-1/

40. World Wide Web Consortium, "Document Object Model (DOM) Level 2 Specification," W3C Recommendation, June 2001, available at http://www.w3.org/TR/DOM-Level-2/

41. World Wide Web Consortium, "Extensible Markup Language (XML) 1.0," W3C Recommendation, June 2001, available at http://www.w3.org/TR/1998/REC-xml-19980210

42. World Wide Web Consortium, "Extensible Stylesheet Language (XSL)," Candidate Recommendation, June 2001, available at http://www.w3.org/TR/xsl/

43. World Wide Web Consortium, "The Query Languages Workshop," June 2001, available at http://www.w3.org/TandS/QL/QL98/

44. World Wide Web Consortium, "XML Accessibility Guidelines," W3C Working Draft, June 2001, available at http://www.w3.org/WAI/PF/xmlgl

45. World Wide Web Consortium, "XML Path Language (XPath)," W3C Recommendation, June 2001, available at http://www.w3.org/TR/xpath

46. World Wide Web Consortium, "XML Query Requirements," Working Draft, June 2001, available at http://www.w3.org/TR/xmlquery-req

47. World Wide Web Consortium, "XML Schema Part 1: Structures," W3C Recommendation, June 2001, available at http://www.w3.org/TR/xmlschema-1/

48. World Wide Web Consortium, "XML Schema Part 2: Datatypes," W3C Recommendation, June 2001, available at http://www.w3.org/TR/xmlschema-2/

49. World Wide Web Consortium, "XQuery: A Query Language for XML," Working Draft, June 2001, available at http://www.w3.org/TR/xquery/

50. XML-QL : A Query Language for XML, Version 0.9, June 2001, available at http://www.research.att.com/~mff/xmlql/

51. XMLShark, developed by infoShark, June 2001, available at http://www.infoshark.com/products/index.shtml

BIOGRAPHICAL SKETCH

Rajesh Kanna was born in Chennai, India. He received his BE degree in computer engineering from the National Institute of Engineering, University of Mysore, Mysore, India in 1998. In 1999, Rajesh was admitted into the CISE graduate program at the University of Florida. He graduated in August 2001 with the Master of Science degree. His research interests include XML-based management systems, data warehousing systems, and database and World Wide Web technologies.