IMPLEMENTING UPDATE EXTENSIONS TO XQUERY 1.0

By

GARGI M. SUR

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

2003

To my parents, Mahadeb and Rina Sur

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor, Dr. Joachim Hammer, for his invaluable guidance throughout my master's studies. With his numerous insights and excellent suggestions, he has always motivated me to pursue my thesis research in a disciplined and dedicated manner. I would also like to express my gratitude to Jérôme Siméon, for giving me an opportunity to use Galax for the prototype implementation; and for his collaborative efforts that were instrumental to the successful completion of this work.

I take this opportunity to thank Dr. Abdelselam "Sumi" Helal, the co-sponsor of the UbiData project, for his valuable comments, advice, and corrections. I also wish to thank Dr. Stanley Su for serving on my supervisory committee.

I would like to thank all my friends, especially Poonam, Ketan, Nikunj, Thrity and Satish for their constant encouragement and support that made the completion of this thesis possible. Above all, I would like to thank my parents for their unending love, affection, and inspiration to strive for perfection and excellence in all my pursuits.

TABLE OF CONTENTS

page

LIST OF TABLES

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

IMPLEMENTATING UPDATE EXTENSIONS TO XQUERY 1.0

By

Gargi M. Sur

August 2003

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

In recent years, XML has emerged as the de facto standard for processing and exchange of information among heterogeneous systems. For XML to gain universal acceptance as a data representation and sharing format, standard techniques must be developed to query and update data stored as XML. The World Wide Web Consortium (W3C) is in the process of formalizing a query language for XML, called XQuery. But the language currently lacks support for updates. Absence of a standard update language has forced many vendors offering XML data management tools to develop their own proprietary XML modification solutions. Significant research is being carried out to improve XML querying and indexing techniques, but updates have received little attention so far.

This thesis deals with techniques to incorporate update extensions into the W3C XQuery standard. An update processor for XML was developed that is tightly integrated with XQuery 1.0, exploiting its powerful syntax and carefully formulated semantics. The prototype is fully functional and executes update processing instructions for the UbiData

mobile profile server. The integrated XML query/update language facilitates application-transparent profile data access and synchronization in mobile environments, which are shown using UbiData use cases in the thesis.

Update processing involves analyzing the top-level update queries, translating them into primitive data modification operations, and executing these operations atomically on the XML data model instance documents. The basic data modification operations (namely, insert, delete, and replace) were implemented as data model primitives. The update queries are parsed, normalized, and evaluated by invoking the primitive data model functions. To ensure consistency of updated data, snapshot semantics were enforced. The prototype implementation was carried out of top the Galax XQuery 1.0 query processor from Bell Laboratories. The modifications and functional extensions to Galax for implementation of data model primitives, the framework for update processing, and various algorithms to evaluate insert, delete, replace, and complex updates in XML are categorically explained in various chapters in the thesis. Finally, several experiments leveraging fast, efficient updates on profile metadata and XML Benchmark documents are described.

CHAPTER 1
INTRODUCTION

Extensible Markup Language (XML)

XML [1] has emerged as the key technology for document processing, messaging, cross-platform communication, and data exchange. It originated as a text-based markup language that separates document content from its presentation. It was designed such that highly specialized user groups could create their own markup languages to meet their requirements more quickly, logically and efficiently. This has spawned the development of XML-based variants such as CellML (for biological models), MathML (for mathematical expressions), GML (for geographical information) etc. that are tailored to meet the needs of various scientific and business applications.

The concept of extensibility provided by XML is not entirely new. Standard Generalized Markup Language (SGML) [2], the precursor to XML, provides similar if not greater functionalities and has been in use for over a decade. SGML is an international standard that was developed primarily for the purposes of processing complex technical documentation and government applications (for example, medical records, the IRS, Department of Defense, large company databases, aircraft parts and library catalogs). However XML was designed to deliver structured content over the Internet and not to replace SGML. The design objective was to provide an easy-to-write, easy-to-interpret, and easy-to-implement subset of SGML. A document encoded in XML can be validated for correctness and constraints using a Document Type Definition

(DTD, a concept borrowed from SGML); or a Schema[1]. Briefly, XML is a strict (but simplified) subset of SGML, offering

- **Extensibility**. Define new elements, containers, attribute names
- **Structure**. A DTD can constrain the information model of a document
- **Validation**. Every document can be validated with respect to a DTD or schema

Later generations of XML standards (such as XML Schema) set the stage for using XML for document and data modeling. To cope with its increasing popularity, most vendors of data management tools now offer XML-based solutions for exporting, viewing, and publishing data. For example, relational database systems from IBM, Oracle, and Microsoft provide support for XML. Several data repositories, like Software AG's Tamino [3] and Progress Software's ObjectStore [4] are now available that store XML in *native* format.

Introduction to XML Query Language (XQuery)

As more and more data is being moved to the XML-based systems, the need to query, retrieve, and regenerate new XML content becomes of primary importance. XQuery 1.0 [5] is the W3C recommended standard to query XML. It is a declarative, functional language that can be used to query both document and database systems. It provides powerful constructs like path and FLWOR expressions, which can be used to formulate any desired query expression. The standard is still in its evolutionary stages. The Query Language Working Group (comprising of representatives from the industry, academia, and the research community) is in the process of formalizing the language. The XQuery specification suite consists of a complex set of *nine* working drafts (Table 1-1).

[1] Schemas are written using XML Schema, which is a W3C Recommendation for describing and constraining the content of XML documents.

Table 1-1. The XQuery specification suite

Specification	Description
XQuery 1.0: An XML Query Language	Description of XQuery syntax and language capabilities
XML Query Requirements	List of XQuery desiderata
XML Query Use Cases	Real world scenarios and XQuery snippets
XML Syntax for XQuery (XQueryX) 1.0	Machine readable XML-based query syntax
XML Path Language (XPath) 2.0	Core language to navigate hierarchy of an XML document
XQuery 1.0 and XPath 2.0 Data Model	Description of data model used to represent the content of XML documents.
XQuery 1.0 and XPath 2.0 Data Model Serialization	Serialization of data model values to XML
XQuery 1.0 and XPath 2.0 Formal Semantics	Underlying algebra specifying the static and dynamic semantics of XQuery expressions
Functions and Operators	Basic functions and operators defined for the language with argument and result types

The XQuery 1.0 document provides a holistic view of the surface syntax of the query language. The use cases and requirements documents present remarkable examples of its usage. XPath [6] provides a compact, URL-like syntax to directly navigate through the hierarchy of an XML document and XPath expressions can be embedded into XQuery. Both XQuery and XPath share a common data model. The Data Model [7] and Formal Semantics [8] working drafts are of utmost importance to the query-engine implementers as they describe the detailed query algebra, the mapping from surface syntax to the underlying algebra, and the static and dynamic query evaluation semantics.

Need for an XML Update Language

So overall, the XQuery specification suite provides an astonishingly well-formulated query language. Many vendors have adopted the language to build query engines based on the standard. But, *the language is not complete*. As a query language, it has powerful constructs to retrieve information and reconstruct new content. But there are no features available to change, manipulate, or modify existing content. *XQuery 1.0 lacks update semantics*. This missing functionality is detrimental to its universal acceptance.

The Query Language Working Group is aware of this shortcoming in the language. Update capabilities are necessary not only to modify existing XML documents but also to manage native XML repositories or XML data published from relational sources. However, there is no agreed upon XML update language and there is very little experience in building an XML update processor. As a result, existing XML storage engines have little support for XML updates, and often rely on ad hoc solutions. Also, the lack of easily accessible XML update infrastructure makes research about the impact of updates on XML processing difficult to carry out.

This thesis describes an update language for XML, its syntax, and its implementation. The update processor for XML was developed as an extension to an existing XQuery processor, named **Galax** [9]. Galax was developed by researchers at Bell Laboratories and AT&T Laboratories. It is a lightweight, portable implementation of the XQuery 1.0 with full compliance to the W3C working drafts (outlined in Table 1-1). The design and implementation was carried out through collaborative efforts between research groups at Bell Laboratories and University of Florida (UF).

The UbiData [10] mobile computing project at UF was used as a test bed for the development efforts. UbiData aims to develop techniques to allow users to seamlessly access their data in mobile environments. UbiData uses user profiles (UP) to manage user identities. These user profiles are XML documents which contain information about the user demographics (e.g., name, identity, password, etc.), descriptions of his/her mobile devices currently in use including their capabilities (O/S type, RAM, disk space, etc.), and information about the most frequently used files owned by the user including access frequencies, whether they are shared or not, etc.

Efficient management of UbiData user profiles requires both query (find a given user's profile, find the preferences for a given list of devices, etc.) and updates (add a new user profile, modify preferences, etc.). UbiData uses XQuery and the proposed update extension for both access and modification to its user profile information. The user profiles are stored in a centralized profile server in the form of XML documents. The update processor, as a component of the profile management server, executes all the profile-based queries and updates. Thus, UbiData proved to be an excellent test bed for the XML update processor.

Contributions

The need for concurrent XML queries and updates is in fact not specific to the user profile scenario in UbiData, but also arises in a number of other application domains, such as in Web services (e.g., an on-line travel agent must support both access to flight information and changes to reservations), in the context of the Semantic Web (to query and maintain an ontology written in RDF), in XML messaging (e.g., to route existing customer transactions or add annotations to them), or in XML publishing (e.g., to update an XML view over an underlying relational store). We believe that the update language, described in this thesis, can be used all such application domains to support querying and updating XML data efficiently.

Briefly, the important contributions of this thesis can be summarized as follows:

- Use of an XML updates language for user profile management in UbiData.
- Prototype implementation of the update language for XML that is tightly integrated with XQuery.
- Evaluation of update processing framework using various evaluation criteria.

The organization of the subsequent chapters of the thesis is as follows: in the next chapter, the state-of-the-art in XML update processing and user profile frameworks is described; the system architecture of UbiData and profile management using XQuery is detailed in Chapter 3. The syntax and semantics of XML update language is explained in Chapter 4. The processing model and execution of updates is explained in Chapter 5. Finally, the evaluation of the update processor using different benchmarks has been described in Chapter 6.

CHAPTER 2
RELATED RESEARCH

Research related to the work described in this thesis falls into the following three areas: XQuery, update proposals for XQuery, and user profile applications, which are described in the following sections.

Overview of XQuery 1.0

XQuery 1.0 is a strongly-typed, declarative, functional language and provides SQL-like constructs for specifying queries. A query module consists of a Query Prolog (which is optional) and a Query Body. All the declarations and definitions that create the environment for query processing, such as namespace declarations, schema imports, module imports, variable and function definitions, etc., are specified in the Query Prolog. The prolog is followed by the query body, which can be evaluated and its value is the result of the query.

XQuery Data Model

The XQuery data model is used to describe the abstract, logical structure of an XML document and the data items a query implementation must understand. Since XQuery is a functional language, every query construct is an **expression**. The value of an expression is always a **sequence**, which is an ordered collection of zero or more items. An **item** is either an atomic value or a node. An **atomic value** is a value in the value space of an XML Schema atomic type (a simple type). The XQuery data model defines seven different kinds of **nodes**: *document*, *element*, *attribute*, *namespace*, *comment*,

processing instruction and *text*. Each node in an XML document instance is assigned a unique identifier by the query engine during query evaluation.

Node Constructors

XQuery provides constructors that can create new XML structures within a query. The simplest way to generate a new element is to embed the element directly in a query using XML notation. Consider the following XML fragment:

```
<item_tuple>
        {$i/itemno}
        {$i/description}
</item_tuple>
```

In this example, the item-tuple element is a valid *element constructor* query, with the caveat that $i is a global variable bound to a valid expression in the Query Prolog.

Path Expressions

XQuery uses XPath expressions for selecting nodes. XPath uses a path-like notation as in URLs for navigating through the hierarchical structure of an XML document. A path expression consists of a series of "steps", separated by "/" or "//", which are evaluated from left to right. The result of each step is a sequence of nodes in document order that serves as a starting point for the next step. Each step can apply one or more predicates to eliminate nodes that fail to satisfy a given condition. Consider the following query expression:

```
document("bib.xml")//book[@year = "1994"]/title
```

The above query is a valid path expression consisting of three steps. The first step locates the root node of a document. The second step contains the "descendant" axis, which locates all the *book* sub-elements, and applies the predicate to select the books published in year 1994. The third step is the "child" axis that retrieves the *title* of every selected book.

FLWOR Expressions

FLWOR expression is the most powerful construct in the language. It supports iteration over a sequence of variable bindings. The name FLWOR, pronounced “flower”, is derived from the keywords: FOR, LET, WHERE, ORDER BY and RETURN.

- The **for** and **let** clauses in a FLWOR expression generate a sequence of tuples of bound variables, which is called the *tuple stream*.
- The **where** clause serves to filter the tuple stream, retaining some tuples and discarding others based on the given condition.
- The **order by** clause imposes an ordering on the tuple stream.
- The **return** clause constructs the result of the FLWOR expression.

The return clause is evaluated once for every tuple in the tuple stream, after filtering by the where clause, using the variable bindings in the respective tuples. The result of the FLWOR expression is an ordered sequence containing the concatenated results of these evaluations. Consider the following FLWOR expression:

```
for $b in document(“bib.xml”)//book
where ($b/publisher = "Addison-Wesley" and $b/@year > 1991)
order by $b/@year descending
    return <book year="{$b/@year}">{$b/title}</book>
```

This FLWOR expression lists the titles of books published by Addison Wesley after 1991 in descending order.

Other Features

Additionally the language has more features like arithmetic, conditional and sequencial expressions, functions, modules, etc. For more details, see the XQuery working draft [5].

Research on XML Updates

XML update is a relatively new area of research that has not been explored extensively. There were a few initial proposals to support updates for XQuery but those

were not adopted in the current W3C working draft. We surveyed the available proposals and have briefly described them in the following sections.

XUpdate Specification

XUpdate [11] was the first specification available on XML updates. It was introduced by *XML:DB Initiative Group* [12] in November 2000 in order to develop a standard API for XML databases. The specification describes the goals and requirements for an XML Update Language and identifies the following design principles:

- **Scope.** The language must describe how to query and update XML content.
- **Simplicity.** The language should be simple, powerful and straightforward to the author.
- **Conformance.** The language specification should be conformant with other XML specifications, such as XML Namespaces, XPath, XPointer, etc.
- **Extensibility.** The language must be open to future extensibility.
- **Format.** The update specification must be an XML element.
- **Object & Parser Model.** The language must be independent of an XML object or parser model like DOM/SAX.
- **Target.** The language must be able to perform to a part or totality of one or more XML documents.
- **Integration.** The language development group must interact with other XML:DB working groups, particularly XML Database API working group.

XML:DB Lexus is a reference implementation of the XUpdate language. A noteworthy feature of XUpdate is that every update is expressed as a well-formed XML element. Unlike XQuery, there is no specific user-level syntax. For example, the XML element below represents an XML update command:

```
<xupdate:append select="/addresses" child="last()">
    <xupdate:element name="address">
       <town>San Francisco</town>
    </xupdate:element>
</xupdate:append>
```

The update element appends the new *address* element as a last child of *addresses* yielding the following XML result:

```
<addresses>
    <address>
       <town>Los Angeles</town>
    </address>
    <address>
       <town>San Francisco</town>
    </address>
</addresses>
```

The main drawback of XUpdate is that it is based on XPath, and not XQuery. XPath being the primary query language, the language does not incorporate the powerful query constructs provided by XQuery like iterative, assignment and conditional constructs.

Updating XML Stored in Relations

In [13], Tatarinov, Ives, Halevy and Weld describe a mechanism for expressing updates on XML structures that are stored in an underlying relational database. They propose update extensions to XQuery along with a set of primitive operations for modifying the structure and content of an XML document. They map XML document into relations using the *Shared Inlining Technique* [14]. Informally, the method tries to cluster parent and child elements together; it "inlines" a child element or makes it an attribute of the parent if there is 1:1 relationship between them otherwise other it creates a new relation for every 1:n parent-child relationship. The challenge lies in translating the XQuery-based updates into SQL statements. Ideally, a single XML update should result in a single SQL command. But a single SQL statement can modify only a single relation. So every update is mapped to one or more SQL statements, which are issued to update an XML document at several levels of hierarchy. Various insertion and deletion techniques using triggers and optimizations like Access Support Relations are described in the paper.

Their approach is significantly different from the update processor implementation described in this thesis. They use relational techniques to store and modify XML documents, whereas we have used a purely XQuery-based processing model to implement the updates extensions. Our update processor can interface with both relational and native XML repositories. We assume that the overhead of creation of XML storage structures and maintenance of hierarchical element relationships is handled by underlying storage manager and the update processor can invoke various operations through a well-defined API.

Patrick Lehti's Data Manipulation Processor for XML

A proposal for XML update syntax was submitted to the W3C Query Group by Patrick Lehti [15]. He implemented a data modification processor for XML that supports updates of type: INSERT, DELETE, REPLACE and RENAME, using compositional, conditional and FLWOR expressions. His update processor mainly consists of two components: a query compiler that takes XQuery based updates and generates XQueryX [16] (formerly known as ABQL) queries; and a QuiP [17] query executor that parses the XQueryX syntax, reads the XML documents, executes the queries on the data, and saves the results. In our implementation, XQuery based update statements are parsed and normalized to XQuery core expressions, which may be type-checked or optimized and directly executed on the input data model. His thesis also describes conflict situations that may arise during the execution of compositional updates (e.g. a sequence of updates consisting of insert and delete statements). We avoid the conflict situations by enforcing snapshot semantics during the execution of sequential updates. Snapshot semantics help to maintain the consistency of updated results.

Microsoft XML Update Proposal

The second proposal for an XML Data Modification Language (DML) was presented to the W3C XQuery Working Group by Microsoft [18] in May 2002. The proposal outlines design principles of the DML based on the principles of XQuery language. It allows the DML constructs to integrate with and make use of the XQuery iterator, assignment and conditional (i.e. FOR, LET and WHERE) constructs. The proposal categorizes the update operations as follows:

- Node-based operations that change node identities (INSERT, DELETE)
- Node-based operations that preserve node identities (MOVE)
- Value-based operations (UPDATE)

The proposal describes the syntax and semantics of basic data modification operations like insert, delete, move, update, replace, and multi-statement DML actions. The proposal introduces the notion of a "statement", which is not included in the XQuery specification. XQuery being a functional language, every XQuery construct is an expression and an expression can have a value. Evaluating an expression does not have side-effects but evaluating a statement can have side-effects, though it may not have a value. A review of this proposal provided a great deal of insight into understanding the semantics requirements of the update language. Some of the ideas presented have been included in the thesis implementation, after some refinements to the original concepts.

Having reviewed the state-of-the-art in XML update processing, the main features of the various update proposals/implementation have been summarized in Table 2-1. Update operations that were proposed by various implementers are: insert, delete, append, replace, rename, and move. Of these, the *insert*, *delete* and *replace* are the basic operations and the remaining ones can be derived from these three functions. We

implemented these operations in the form of *data model primitives* and provide a public API to interface them with the underlying storage model. Based on the aforementioned proposals, a working draft for XML updates was compiled by the members of the W3C Query Working Group (which will be published at a future date). Our update processor implements the language extensions to XQuery described in this working draft. The update language syntax and semantics are explained in detail with examples based on UbiData user profiles in Chapter 4.

Table 2-1. Comparison of XQuery update proposals

	XUpdate	Updating XML in Relations	Lehti's Proposal	Microsoft Proposal
Basic Update Operations	Insert-before Insert-after Append Update Remove Rename	Insert Insert-before Insert-after Delete Rename Replace	Insert Delete Replace Rename	Insert Delete Move Update Replace IF DML FLW DML
Query Language	XPath	XQuery	XQuery	XQuery
Scope of updates	Elements Attributes Text Comments PI	Elements Attributes Entity references	Elements Attributes Text Comments PI	Entire document except metadata attributes
Implementation	XML:DB Lexus 0.3	Implemented using IBM DB2	XQueryX parser + Quip processor	NA
Additional Features	Each update is an XML fragment; specifically "xupdate:modi-fications" element is used to indicate update elements.	Storage of XML data using the "Shared Inlining" technique. Outer Union Method for XML results Trigger and ASR based updates	Conflicts in compositional updates Sorting of core updates to remove conflicts Type-checking implemented	Introduces the notion of update statement. Multi -statement DML actions (if & FLW clauses) Well-defined semantics for execution of DML actions

Research on User Profiles

User profiles are mostly used for identity management in multi-user systems. For examples, wireless systems use profiles for authenticating the caller, locating the callee, fast call routing, updating billing records, etc. The user profiles contain data such as user biographical information, user preferences, per user basis service customization data, service usage statistics, payment information, etc. The main advantage of using profiles is that user relevant data can be categorically stored in database management systems and queried easily. User profiles have been introduced in a vast variety of applications and software systems. However, different applications and systems generate profiles according to their requirements and utility factors. Listed below are some common applications or systems that use profiles:

- The Microsoft Windows servers store NT domain user profiles which can be configured for single users or group of users.
- Netscape browsers use roaming profiles, which can be used to access the fully user-personalized *Communicator* environments from remote machines. A profile (preferences, bookmarks, address book, security files, and so on) is stored in a central repository on a profile server. When a user with roaming profile settings logs in, the profile is retrieved from the server and transferred back when the user quits.
- Microsoft started the Passport authentication system in 2001, later known as .NET Passport Service, to provide single sign-in services based on user profiles that enables users to visit other Web sites and access password-protected services, such as instant messaging and online banking, without having to sign in at those sites. Sites which currently use the sign-on services include Starbucks.com and Microsoft's own MSN Internet sites.

Need for interoperable services and systems have led to the emergence of several profile-sharing frameworks. The Liberty Alliance Project, 3GPP and GUPster frameworks are three such exemplary architectures and have been described in further sections.

Liberty Alliance Federated Identity Management

Liberty Alliance Project [19] was formed in 2001 to develop architecture and protocols for advanced identity management on the Internet. User identities are scattered across numerous entities (e.g. banks, credit card companies, brokerage firms, insurance companies, national IDs, pension funds, medical providers, and workplaces). To avoid replication of identities and to ensure ease-of-use, rapid access to distributed identities; the alliance advocates sharing of user identities amongst individuals and businesses in an open, *federated* way, respecting the privacy and security of shared identity information.

Users benefit with a simplified sign-on service that grants them access to resources to which they have permissions. Businesses benefit by conducting transactions with authenticated employees, customers, and partners. The key features of the Liberty Alliance architecture are permission-based attribute sharing, an identity discovery service, and exchange of security profiles based on Security Assertions Markup Language (SAML) protocols.

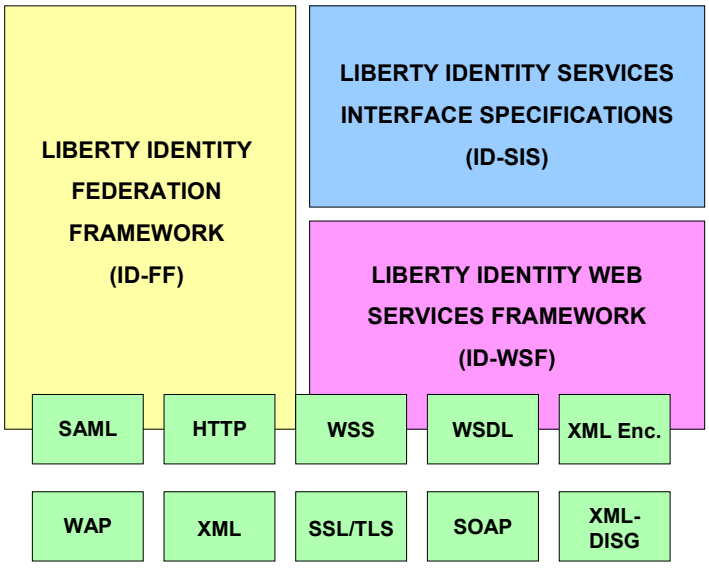

Figure 2-1. High-level overview of Liberty Alliance architecture

- The Liberty Identity Federation Framework (ID-FF) enables identity federation and management through features like identity/account linkage, simplified sign-on, and simple session management.

- The Liberty Identity Web Services Framework (ID-WSF) provides the framework for building interoperable identity services, permission based attribute sharing, identity description and discovery, and the associated security profiles.

- The Liberty Identity Service Interface Specifications (ID-SIS) provides the schema and instantiation of the technical implementation as defined in the ID-WSF to provide for interoperable identity services.

The Liberty Identity Personal Profile (ID-PP), included in the ID-SIS, comprises the profile specification. ID-PP is a Web Service, hosted by an application provider and discovered via discovery service. It offers:

- Basic profile information
- Contact information
- Basic demographics
- Presentation information
- Employment details
- Extensions for arbitrary data

A typical principal has two ID-PP instances, one for her work identity, and another for her private identity. The ID-PP attribute data can be queried and updated using mechanisms for access control and data validation.

The 3GPP Generic User Profile (GUP)

The 3rd Generation Partnership Project (3GPP) is a collaborative agreement established in 1998 to develop a complete set of globally applicable Technical Specifications and Reports (TSR) for a 3G System. One of its objectives is to develop a Generic User Profile (GUP) that provides a generic mechanism to access and manipulate user related information. The GUP specification [20] describes a generic user profile as "the collection of user related data, which affects the way in which an individual user experiences services and which may be accessed in a standardized manner." The Generic

User Profile is being defined using XML. Using GUP, user data can be retrieved and managed in a uniform way. However the data content itself is not described within the Generic User Profile, but only the data model and schema are defined. The user profile for each user is said to consist of several components. A general outline of the various components is as follows:

- Authorized and subscribed services information, e.g., service descriptions
- General user information, e.g., settings, preferences, phone books and buddy lists
- PLMN specific user information, e.g., addresses and WAP/GPRS parameters
- Privacy control data, e.g., privacy settings for standardized services
- Service specific information, e.g., customization data, keys and certificates
- Terminal related data, e.g., UI capabilities and configuration details
- Charging and billing related data, e.g., billing policies and intervals

The GUP does not recommend storing run time, session, call or application execution, and historic/statistical data as a part of the profile. A generic feature of the user profile is that different entities are data consumers for a certain subset of the user profile and are data suppliers for another part. The 3GPP GUP data is distributed in nature and consequently stored in the user environment, home network, and value added service provider equipment.

GUP-based Framework for Converged Networks

GUP^{ster} [21] is a data management framework to facilitate sharing profile data for converged services. Interoperability between voice, telephony, internet and other networks has led to the emergence of converged networks. Providing interoperable services across such networks again requires a broad array of information about end-users, including, awareness of presence, location, and availability of devices, billing models and plans, and support for highly personalized renditions of services, which entail storage and access of large amounts of personal data or profiles. GUP^{ster} combines ideas

from the standard schema of 3GPP GUP, federated architectures for data integration, and the Napster approach for peer-to-peer sharing to the specialized context of profile data management. In GUP^{ster}, data stores that are willing to share user profile components join the GUP^{ster} community by registering their components on the GUP^{ster} server. Thus user profile information is distributed but the meta-data is centralized and the GUP^{ster} server stores and provides referrals to the data stores and profile components.

The survey of various user profile applications was helpful to develop the structure and organization of user profiles for UbiData. UbiData profiles (UP) have similar structural components compared to the 3GPP GUP and ID-PP profiles. In a UP, the user demographics, preferences, privacy, and personal data components have been categorized separately. A hierarchical structural organization of the UP schema makes it feasible to incorporate the UP into the profile management frameworks described earlier. For example, the UP can be adapted as a data source in the GUP^{ster} framework by creating wrappers that interface it with GUP^{ster} profile server.

We present the organization of the UP and profile management in UbiData using XQuery in the following chapter.

CHAPTER 3
PROFILE MANAGEMENT IN UBIDATA

UbiData [10, 22] is a mobile computing research project[2] being carried out at the University of Florida. UbiData aims to develop techniques to allow mobile users to ubiquitously access their data in mobile environments. Many difficulties arise while attempting to provide rapid access to user data due to severe mobile network and device constraints. The challenges imposed by mobility as identified by UbiData are:

- Any-time, any-where access to data; regardless of whether the user is connected, weakly connected via a high latency, low bandwidth network, or completely disconnected.
- Device-independent access to data; where the user is allowed to use and switch among different portables even while mobile.
- Mobile access to heterogeneous data sources; such as files belonging to different file systems and/or platforms.

To meet the above challenges, the strategy of optimistic data replication [23] has been chosen in UbiData. The project goal is to develop optimistic schemes for smart selection, hoarding, replication, and synchronization to alleviate the difficulty of data access in adverse mobile environments.

User Profiles and Their Relevance to UbiData

Efficient data management in UbiData entails fast response times with high throughput in the presence of varying connectivity. Exclusive research [24, 25] to develop effective schemes based on user calling and mobility patterns have identified databases as the critical component for data management protocols.

[2] The research was partially funded by the National Science Foundation under grant number CCR-0100770.

In UbiData, the user data is identified by the system and stored in a centralized data warehouse. The initial development efforts using data warehousing technology led to an important observation: *data access is primarily user-centric.* The system has to store and manage huge quantities of data pertaining to its users. The data stored in the central server can be organized into *System Metadata* and *User Data*. Information pertinent to user access and behavior can be grouped as System Metadata, for example, user registration, device configurations, file descriptors, hoarding set, etc., which is partially structured or semi-structured in nature. The User Data comprises of files and their replicas which exist in a vast variety of forms (structured, semi-structured and unstructured) and formats (text, binary, multimedia, markup, etc.). To deal with variance in structure and format of data, there was a need for data representation technology that caters to storage and manipulation of both system metadata and user data. XML, with its inherent features like flexibility and extensibility, is the ideal language to store and exchange data in UbiData or any other similar ubiquitous computing environment.

In addition to the data representation model, the organization of data also plays a dominant role in system performance. *User profiles* are often used to store user-related information in wireless networks in a structured manner. They contain authentication, home-location, visitor-location, call waiting, address book, user preferences, and other such myriad information by the service providers. We adopted the profile-based data organization model [26] to store user relevant information in UbiData. Thus, System Metadata is organized in the form of UbiData user profiles (UP). The profiles are stored as XML documents, and accessed using XQuery. The profile-based system architecture, UP requirements, and XQuery access methods are explained in the following sections.

System Architecture and Components

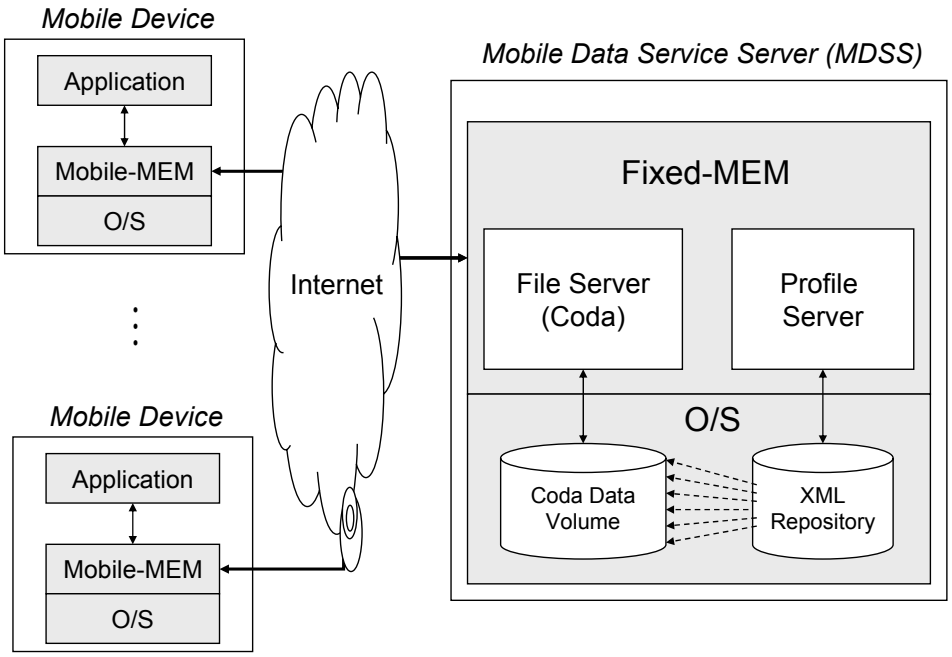

Figure 3-1. Conceptual overview of UbiData system architecture.

The overall system architecture is shown in Figure 3-1. The Mobile Data Service Server (MDSS) provides a highly-available data service for any mobile client. UbiData uses a middleware component called the Mobile Environment Manager (MEM) to adapt to the mobile environment. The MEM is split into separate units Mobile-MEM (M-MEM) and Fixed-MEM (F-MEM) based on varying functional requirements. The M-MEM operates in every mobile host and the F-MEM operates in the mobile server. M-MEM and F-MEM cooperate to provide the mobile data selection, replication, consistency control and network adaptation. They communicate through a set of XML based protocols. The request or reply data is encoded in XML and shipped over the Internet using an asynchronous, reliable message system, like HTTP or SMTP.

To automate the manual process of data-hoarding, the M-MEM needs to determine which files are of importance to the user and cache them in the MDSS. The set of files which are frequently accessed by a user constitutes the user's *working set* (UWS). The data in the user working set is managed by Coda [27], which forms a part of the mobile file system. On each mobile device, M-MEM monitors the user's file access patterns to determine the active working set. Uninteresting or system files are filtered out with the help of exclusive filters. The M-MEM propagates files accesses and updates to the F-MEM, which initiates synchronization procedures with the copy on the MDSS. If the data item is new and does not have a counterpart on the MDS, a file descriptor entry is made in the metadata and a copy is created on the MDSS. Otherwise, the previous copy is modified and the metadata is updated. If immediate synchronization is not desired or not possible (e.g., no connection can be established), updates to the working set are performed upon return to the docking station.

The MDSS uses two distinct repositories to store user data and profile metadata. The metadata server is currently a native XML repository called Xindice [28]. All the relevant meta-information for each UbiData user is captured in an UbiData Profile (UP). When a user joins UbiData, a new profile is created which contains user demographic details and device ownership information. The file usage and user working set information is subsequently collected by the M-MEM and inserted in the UP by the F-MEM. The UPs are stored in the *profile repository* on the MDSS, along with default rules for device capabilities and data conversion filters to support transmittal between different devices. The requirements, description, and schema of the UP are described in the following sections.

User Profile Requirements

The requirements for UbiData profile structure and organization are as follows:

- User demographic information, such as, name, address, devices, etc., must be a part of the profile.
- File access patterns and user working set which is determined by the F-MEM must be a part of the profile.
- All user device information must be captured by the UP.
- The user should be allowed to override the system defined parameters in the profile.
- User preferences must be captured by the profile.
- If data files are replicated on different devices owned by the user, all replica information must be captured by the profile.
- File sharing amongst multiple users must be allowed.
- Queries and updates to the UP should be possible.
- Synchronization parameters, like update frequencies, can be a part of the profile.

UP Schema and Description

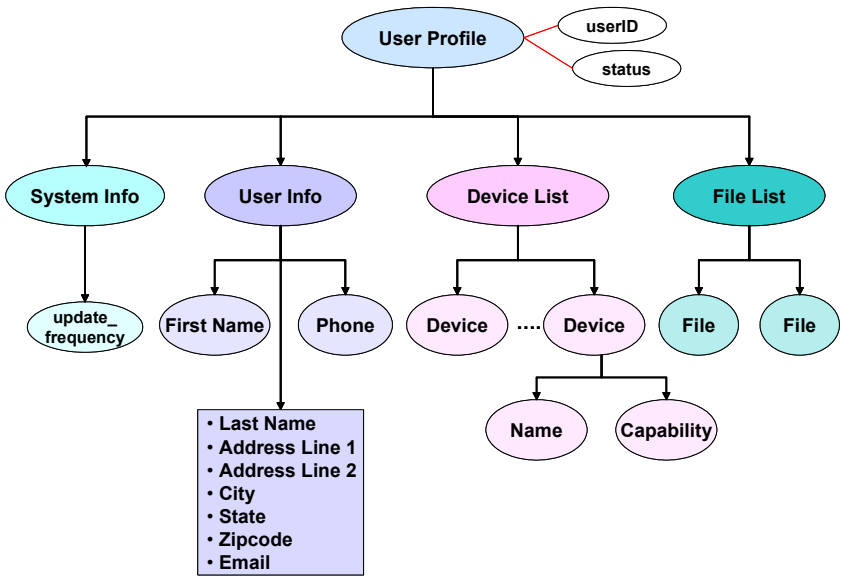

Figure 3-2. User profile schema.

The schema of the UP is shown in Figure 3-2. Every UP has a unique "userID" attribute to identify and access the profile. The "status" attribute captures the state information of the profile, whether it is a newly created one, or is actively being used, or whether the profile has been deleted. A UP consists of four main components:

- System information
- User information
- Device list
- File list

For each component, the UP maintains a list of parameters containing the specific meta-information which has been explained in the following sections.

System Parameters

System parameters specify system-enforced attributes and configuration settings of the middleware such as M-MEM or F-MEM. For example, rather than using a system-enforced frequency for updating the contents of the UP and associated files, a user may opt for a higher update frequency. Meta-information describing system settings are captured by the following two parameters:

- Update-frequency
- State {possible values are: *active, inactive*}

User Parameters

The UP contains demographic details of every user, such as, name, address, email, phone numbers, etc, which are captured by the following parameters:

- First name
- Last name
- Address {Address Line1, Address Line 2, City, State, Zip code}
- Email
- Phone numbers
- Password
- Credit card numbers
- Preferences

Device Parameters

All devices owned by the users are registered with the system and recorded in the UP. Each device has an associated device name, which forms a part of the UbidataURI to specify file locations on mobile devices. The device related parameters are as follows:

- Device name
- Device description
- Device capabilities
- Encryption key

File Parameters

The UP contains files and their replicas that are owned and group-owned (shared) by the user. Files that are "members" of the active working set are specially marked using their *file-status* element. The membership in the working set is predicted based on past user behavior rather than exact knowledge. The M-MEM monitors the user file access pattern and assigns a metric called *hybrid priority* to each accessed file. Hybrid priority serves as an indication of the activeness of files. Files with the highest hybrid priority are hoarded on a user import or synchronize request. The hybrid priority is a function of the three variables, namely recency, frequency and active period:

- **Recency** is the time interval elapsed since the last access and is expressed by the last modification time. This field is not mandatory.
- **Frequency** is the number of accesses by the user of a particular file within the active period. This field is mandatory.
- **Active period** is the time between the first and last access. It can be computed as the difference of last modification time and creation time, it is not mandatory.

Every data file listed in the UP must be uniquely identified using following URI:

```
UbidataURI = mfs://hostname/pathname
```

where *hostname* `= $MobileDataServerRoot | $username/$devicename`

and `mfs` is the mobile file system protocol observed in UbiData.

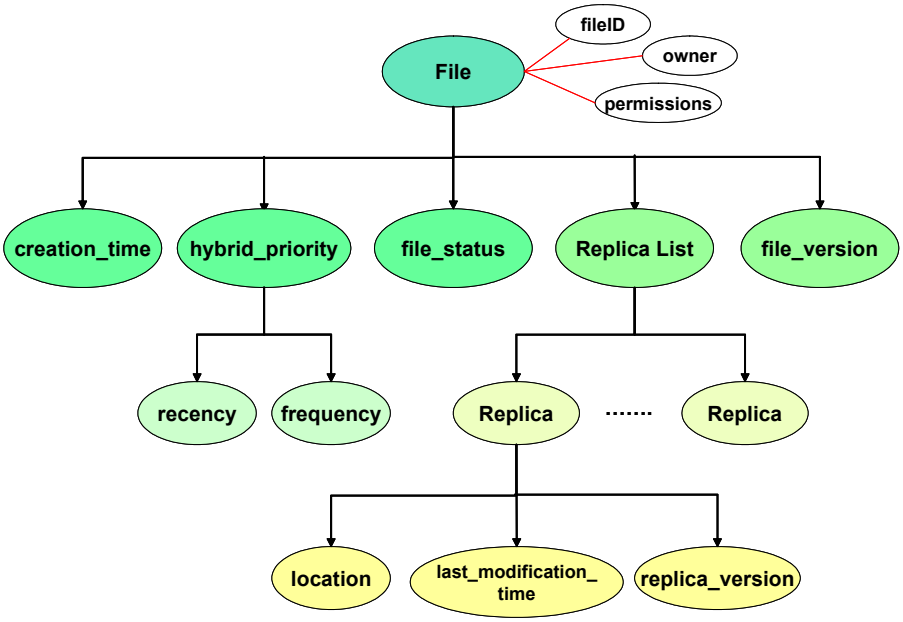

Figure 3-3. File element schema in a user profile.

Thus, file information is described using the following parameters (figure 3-2):

- Filename/file-id
- Owner
- Group-id
- Permissions
- Creation-time
- File-version
- File-status {possible values are: *in-sync, not-in-sync*}
- Hybrid-priority {sub-elements are: *recency, frequency , active-period*}
- Replica list

Each replica list stores details of all replicas of an existing file. For each replica, the following information is stored:

- Location
- Last-modification-time
- Replica-state
- Replica-version

The DTD and schema describing the structure of UP is provided in Appendix A and B respectively. A complete UP instance document is shown in Appendix C.

A simplified sample of UP document is shown in Figure 3-4.

```
<user_profile userID = "gsur" status = "active">
    <system_info>
         <update_frequency>500</update_frequency>
    </system_info>
    <user_info>
        <username>
            <firstname>Gargi</firstname>
            <lastname>Sur</lastname>
        </username>
        <password>Psdf84n87</password>
    </user_info>
    <device_list>
        <device id = "laptop">Dell Inspiron4100</device>
        <device id = "PDA">Compaq iPAQ 3890</device>
    </device_list>
    <file_list>
        <file>mfs://MDS/gsur/docs/todo.txt</file>
        <file>mfs://MDS/gsur/ubidata/arch.doc</file>
    </file_list>
</user_profile>
```

Figure 3-4. Simplified user profile in XML.

User profiles are queried and accessed using XQuery. These queries can be directly executed by the XQuery processor integrated into the profile manager in the F-MEM. The XQuery queries can be also embedded in XML messages exchanged between the M-MEM and F-MEM components. Most queries can be elegantly expressed using FLWOR expressions. The profiles are accessed using the user identifiers and the results are expressed in XML. We present some user profile queries using XQuery 1.0 below using examples.

Query 1: Query the user profile of user "gsur" to retrieve her password.

```
for $u in document("user_profiles.xml")//user_profile[@userID="gsur"]
return
    <keyreply>
       <user>{$u/@userID}</user>
       <password>{$u/user_info/password}</password>
    </keyreply>
```

The result of the above query:

```
<keyreply>
    <user>gsur</user>
    <password>Psdf84n87</password>
</keyreply>
```

Query 2: Retrieve the device list of user “gsur”.

```
for $u in document(“user_profiles.xml")//user_profile[@userID=”gsur”]
return
  <user_device_list>
    <user>{$u/@userID}</user>
    <device_list>
       { for $d in $u/device_list/device
         return $d }
    </device_list>
  </user_device_list>
```

The result of the above query is as follows:

```
<user_device_list>
    <user>gsur</user>
    <device_list>
        <device id = ”laptop”>Dell Inspiron4100</device>
        <device id = ”PDA”>Compaq iPAQ 3890</device>
    </device_list>
</user_device_list>
```

Query 3: Query the user profile of user ‘gsur’ to retrieve the hoarding list, sorted by the file hybrid priority in descending order:

```
for $u in document("user_profiles.xml")//user_profile[@userID=”gsur”]
return
    <user_hoarding_list>
      <user>{data($u/@userID)}</user>
      <hoarding_list>
      {
       for $f in $u/file_list/file
       order by $f/hybrid_priority/@value descending empty least
       return
         <file>
           <name>{data($f/@fileID)}</name>
           <hybrid_priority>{data($f/hybrid_priority/@value)}</hybrid_priority>
         </file>
       }
      </hoarding_list>
    </user_hoarding_list>
```

The result of above query is as follows:

```
<user_hoarding_list>
    <file>
        <name>mfs://MDS/gsur/docs/todo.txt</name>
        <hybrid_priority>8</hybrid_priority>
    </file>
    <file>
        <name>mfs://MDS/gsur/ubidata/arch.doc</name>
        <hybrid_priority>6</hybrid_priority>
    </file>
</user_hoarding_list>
```

XQuery queries suffice whenever some information needs to be retrieved from the UP. But if the M-MEM wants to transmit any changes or updates to the profile, then those updates cannot be directly executed by the XQuery processor on the profile server, as XQuery lacks update functionality. In the next chapter, we present an update language for XML that is tightly integrated with XQuery. The syntax and the semantics of the language are described in detail and explained using examples based on the UP described in Appendix C.

CHAPTER 4
XML UPDATE LANGUAGE: SYNTAX AND SEMANTICS

In this chapter, we present the language extensions for supporting updates in XQuery. The update syntax is explained using the same basic Extended Backus-Naur Form (EBNF) notation as used in the XQuery specification. The language constructs are described in the form of grammar productions.

In order to incorporate the update statement into XQuery, the current QueryBody production of XQuery is extended as follows:

```
QueryBody       :: =    (ExprSequence | Update)?
Update          :: =    SimpleUpdate
                        | ComplexUpdate
SimpleUpdate    :: =    Insert
                        | Delete
                        | Replace
                        | EmptyUpdate
ComplexUpdate   :: =    FLWUpdate
                        | ConditionalUpdate
```

Updates are expressed using statements that extend the XQuery syntax. An update *statement* is different from the traditional XQuery *expression* in that it modifies an existing value instead of simply returning a value. Updates are classified into *simple* and *complex updates*. Simple updates represent the basic data modification operations: add, remove, and replace. Complex updates can be either conditional or iterative, allowing for complex operations based on simple update constructs.

Simple Insert

The simple insert statement supports insertion of a new XML fragment into an existing node. Simple insert preserves the original order of nodes in the document. The syntax of the simple insert is described as follows:

```
Insert          :: =    UpdateContent InsertLocation
InsertLocation  :: =    ((<“as” “last”> | <“as” “first”>)? “into”
                        TargetNode)
                        | (“after” TargetNode)
                        | (“before” TargetNode)
UpdateContent   :: =    Expr
TargetNode      :: =    Expr
```

The insert statement inserts a copy of the item sequence returned by the UpdateContent into the location indicated by InsertLocation. XQuery is used to compute the location where the update occurs and the content of the update. Thus, UpdateContent is evaluated to create *update content*, which may be any sequence of items, and InsertLocation is evaluated to determine the *target node* of the insert. The target node of the insert must be single node (i.e., if the InsertLocation has an empty value or evaluated to contain more than one element, then an error is raised and no insertion is performed).

The syntax allows insertion into the first and last position in an element or document node. It also allows insertion before or after a sibling node, which may be an element, comment or processing instruction node. Attribute nodes in update content are always inserted before the insertion of any element nodes.

The following update inserts a new UP as the last element in the profile repository:

```
insert
    <user_profile userID=”jdoe” status=”new”>
        <user_info>
            <username>
                <first>John</first>
                <last>Doe</last>
            </username>
            <password>XcvgfT35R</password>
        </user_info>
        <device_list>
            <device deviceID=”laptop”>
              <device_name>Sony Laptop</device_name>
              <device_capabilities>15" TFT and ATI Radeon</device_capabilities>
            </device>
        </device_list>
    </user_profile>
as last into
    document(“user_profiles.xml”)/user_profiles
```

The following illustrates a more generic insertion which takes two external variables $id and $name as parameters and inserts the new profile in alphabetical order (using the insert location ‘after’ along with the XPath predicate ‘last()’):

```
insert
  <user_profile userID="{$id}">
    <name>{$name}</name>
  </user_profile>
after
  document("user_profiles.xml")//user_profile[@userID < $id][last()]
```

Simple Delete

The simple delete statement deletes an existing XML fragment from the XML data model instance. The syntax of delete is as follows:

Delete	:: =	“delete” TargetNodes
TargetNodes	:: =	Expr

The delete statement removes all the nodes returned by the TargetNodes expression. Every item in the value returned by evaluation of TargetNodes must be a node, or else an error is raised and no nodes are deleted. When a node is deleted, the elements in the sub-tree rooted at that node must be deleted prior to its deletion.

The following delete statement removes all user profiles which have not been used for more than a year:

```
delete
  document("users.xml")//profile[@modified < (date() - xdt:duration("P1Y"))]
```

Simple Replace

The simple replace statement supports “replacement” of an existing XML fragment with a new one. The syntax for replace is as follows:

Replace	:: =	“replace” <“value” “of”>? TargetNode “with” UpdateContent
TargetNode	:: =	Expr
UpdateContent	::=	Expr

The UpdateContent is evaluated to determine the replace content. The replace statement replaces the target node returned by TargetNode, or the value of the target node (if "value of" clause is specified) with a copy of the item sequence in the replace content.

The following constraints apply for a replace statement:

- The target node must be a single node.
- If the target node is an element node, then the replace content must be a *Content Sequence*. A content sequence is any sequence of zero or more element nodes, atomic values, processing instruction and comment nodes. Moreover, if an atomic value is being inserted, then it must first be converted into a text node.
- If the target node is an attribute node, and if the value of clause is specified then the replace content must be a sequence of zero or more atomic values; other wise if the value of clause is not specified, then the replace content must be a sequence of attribute nodes.
- If the target node is a document node, and if the value of clause is specified then the replace content must be a content sequence; other wise if the value of clause is not specified, then the replace content must be a document node.
- If the target node is a text, processing instruction or comment node, and if the value of clause is not specified then the replace content must be a content sequence; other wise if the value of clause is not specified, error is raised.

The following replace statement updates the last modified time of file "arch.doc" of user "gsur" with a new value:

```
replace value of
  $user_profiles//user_profile[@userID="gsur"]//file[@fileID="mfs://MDS/gsur/
    docs/arch.doc"]/last_modification_time
with
  "Wed Jun 25 4:45:36"
```

Conditional Update

The conditional complex update is built using the traditional if-then-else construct.

The syntax of the conditional update is as follows:

```
ConditionalUpdate    :: =    "update"
                             <"if" "("> ConditionalExpr ")"
                               "then" SimpleUpdate
                               "else" SimpleUpdate
```

A conditional update performs an update depending on a boolean condition. If the *Effective Boolean Value*[3] of ConditionalExpr is true, then the SimpleUpdate in the *then* clause is evaluated; otherwise, the SimpleUpdate in the *else* clause is evaluated.

The following conditional update statement replaces the device-name element in the UP of user *gsur* if the laptop device is owned by the user exists other wise inserts a new device descriptor into the UP:

```
update
  if ($profiles//profile[@userID="gsur"]/device_list/device[@devID="laptop"])
  then
    replace
      $profiles//profile[@userID="gsur"]//device/@devID="laptop"]/device_name
    with
      "IBM T23 Series"
  else
    insert
      <device devID = "laptop">
        <device_name>IBM T23 Series</device_name>
        <devive_capabilities>2GHz Centrino</device_capabilities>
      </device>
    into $user_profiles//user_profile[@userID="gsur"]/device_list
```

Empty Update

The empty statement is written as "( )" and can be used to indicate that no data modification must be performed. This can be useful, for example, inside a conditional update when only one of the branches of the conditional must perform a data modification.

For example, the following update adds a new device to a profile if only one does not already exist:

```
update
  if ($user_profile//user_profile[@userID=$id]/device[@deviceID=$did])
  then
    insert
      <device deviceID="{$did}"/>
    into $user_profiles//user_profile[@userID=$id]
  else
    ()
```

[3] Effective Boolean value is defined as the result of invoking the fn:boolean function on a XQuery sequence.

FLWUpdate

FLWUpdate is a complex update that is built from simple updates using FLWOR expressions. The syntax for FLWUpdate is as follows:

```
FLWUpdate        :: =   "update"
                          (ForClause | LetClause)+ WhereClause?
                            SimpleUpdate+
ForClause        :: =   "for" "$" VarName TypeDeclaration? PositionalVar?
                            "in" ("," "$" VarName TypeDeclaration? PositionalVar? "in" Expr)*
LetClause        :: =   "let" "$" VarName TypeDeclaration? ":=" Expr
                            ("," "$" VarName TypeDeclaration? ":=" Expr)*
TypeDeclaration  :: =   "as" SequenceType
PositionalVar    :: =   "at" "$" VarName
WhereClause      :: =   "where" Expr
```

The *for* and *let* clauses in the FLWUpdate generate sequence of tuples of variable bindings. The *where* clause evaluates the predicate expression and applies the evaluated conditional value to filter the tuple stream. The SequenceType in the TypeDeclaration production is used to specify the type of an XQuery expression, i.e. to describe the value of the item sequence generated by evaluating the expression. The SimpleUpdate is then evaluated once for every tuple in the tuple stream. The simple updates are executed in the same order as implicitly specified by the update statement.

For example, the following FLWUpdate statement increases the synchronization frequencies of all PDA's by 10 percent:

```
update
  for $d in doc(user_profiles.xml)/user_profiles//device_list/device
  let $f := $d/syncfreq
  where $d/@deviceID="PDA"
    replace value of $f
      with  $f * 1.1
```

In the above FLWUpdate, `$d` variable binds to all the device elements in the user profile listings and `$f` binds to their corresponding synchronization frequencies. Then the replace statement is executed once for every binding of `$d` to increment `$f` value by 10 percent.

Snapshot Semantics for FLWUpdates

As seen above, the FLWUpdate contains a list of simple updates which must be executed in order as specified in the update statement. The semantics of these operations need to be carefully defined as they may result in inconsistent results. For example, consider the following FLWUpdate statement that is sent to the F-MEM by a temporarily disconnected M-MEM. The M-MEM, upon reconnection, batches the incremental updates (in this case: simple replace statements) together into a single complex update statement and transmits it to the F-MEM.

```
update
    for $f in $user_profiles/user_profile/file_list/file
    where $f/@fileID="mfs://MDS/gsur/ubidata/usecases.doc" and $f/owner="gsur"
        replace $f/hybrid_priority/frequency with
            <frequency>{$f//frequency + 10}</frequency>
        replace $f/frequency with
            <frequency>{$f//frequency + 15}</frequency>
```

The above FLWUpdate replaces the *frequency* value of *usecases.doc* file owned by user *gsur*. Assume that the initial value of frequency is 5. The result of the frequency element must be computed with respect to the initial value of frequency. When the F-MEM executes the update statement, the two simple replace statements in the body of the FLWUpdate are evaluated serially. The execution can have two possible outcomes:

```
<file owner = "gsur" name = "usecases.d">
    <frequency>30</frequency>
</file>
```

or

```
<file owner = "gsur" name = "usecases.d">
    <frequency>20</frequency>
</file>
```

The first outcome is based on the premise that the result of the first replace is visible to the latter, whereas the second outcome is based on the premise that the two updates executed independently, i.e. the result of the first replace is not visible to the

latter and effect of the second replace overwrites the result of the former one. The question then arises, which outcome is acceptable? The answer depends on the *semantics* that are followed to evaluate the FLWUpdate statements.

Unclear semantics may result in inconsistent results. To ensure consistency in the overall result of the FLWUpdate, *snapshot semantics* are enforced. Snapshot semantics entails that *all the variables in for and let clauses of FLWUpdate must be bound with respect to the initial snapshot before the simple updates in the body of the FLWUpdate are executed*. The simple update statements are then executed sequentially based on the initial snapshot and evaluated independently of each other. So, in our previous FLWUpdate example, the second outcome will be considered correct.

In the next chapter, the implementation of a prototype for the XML update language is described, along with the algorithms for insert, delete, and replace operations using XML tree data model structures, and methods to enforce snapshot semantics for FLWUpdates.

CHAPTER 5
IMPLEMENTATION OF XML UPDATES

The update processor for XML has been implemented on top the Galax XML query processor [29] developed at Bell Laboratories. Galax is an open source implementation of XQuery 1.0 and closely conforms to the XQuery specification suite. Galax implements various features of XQuery like path and FLWOR expressions, arithmetic and comparison operators, node constructors, document order, etc. It also supports several advanced features namely XML Schema import and validation, static type checking, and type/value based optimization. Galax is written in OCaml and is portable to various platforms (Linux, Solaris, Macintosh, and Windows). OCaml [30] is an object-oriented variant of ML with sophisticated features, like typed compilation, automatic memory management, polymorphism and abstraction. It is an ideal language to implement XQuery as well as update extensions, as its algebraic types and higher-order functions help to simplify the symbolic manipulation that is central to query transformation, analysis, and optimization.

Update Processing Model

The update processing model of prototype is based on the XQuery processing model, as described in the XQuery Formal Semantics [8]. The architecture of the update processor is shown in figure 5-1. We implemented update functionality as enhancements to the query processing modules of Galax. The inputs to the update processor consist of update statements and one (or more) XML document(s) that are subject to modification. The various processing stages are parsing, normalization and execution.

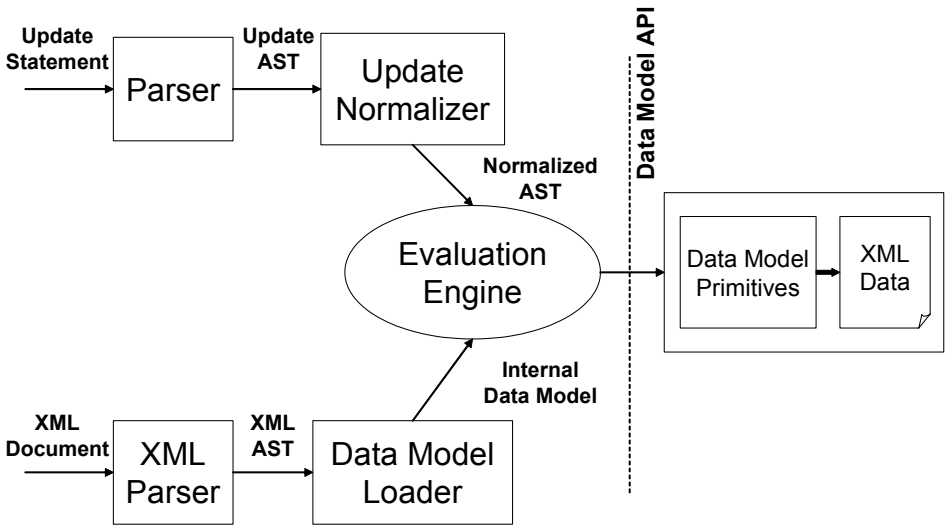

Figure 5-1. Update Processing Model

Parsing

The parser module parses the input update statement and builds an Abstract Syntax Tree (AST) corresponding to the update syntax. The input XML document is processed by a SAX[4] parser. The AST corresponding to the XML document is never materialized. The SAX parser optimizes memory usage by generating stream of SAX events which are consumed by the data model loader to create an instance of the XQuery data model in memory.

Normalization

The update normalizer maps the update AST to an internal representation that can be directly executed by the evaluation engine. It applies XQuery *normalization judgments* (which are described in the XQuery Formal Semantics) to transform the query expressions to equivalent XQuery Core expressions. The normalization judgments are

[4] SAX is an event-based XML processing API

transformations or rewriting rules that expand the abbreviated, hidden, or implied syntax of the top-level language and make them explicit.

FLWUpdate normalization is a special case of update normalization, where in addition to the regular mapping rules; a tuple construction rule needs to be applied. Consider the following FLWUpdate in Figure 5-2.

```
update
    for $u in $user_profiles/user_profile
    where $u/@status = "expired"
        delete $u/file_list
```

Figure 5-2. FLWUpdate statement that deletes ‘expired’ profiles

When this FLWUpdate executes, a *tuple sequence* is generated to hold the initial variable bindings of the "for" and "let" clauses, which are used to evaluate expressions in the simple update list and must satisfy snapshot semantic requirements. This implies that a *tuple* (structure) must be present in the update AST to hold the tuple sequence values.

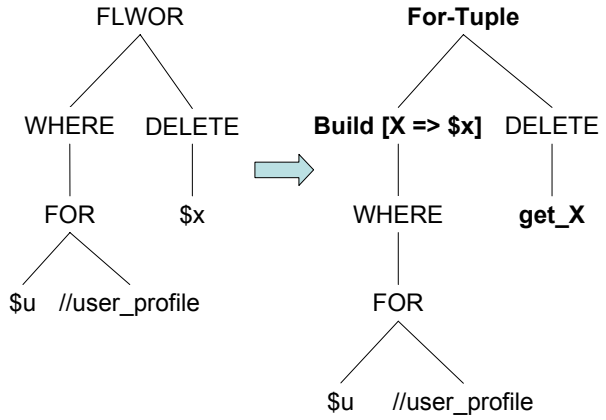

Figure 5-3. Normalization process for FLWUpdate

When the FLWUpdate is normalized, we insert a “for-tuple” construct into the update AST, which is later materialized during execution phase. The normalization process is shown in Figure 5-3. The “build” and “get_X” clauses represent internal tuple construction and extraction operations respectively.

```
update
    for-tuple $dot in
      (for $u in $user_profiles/user_profile
    where $u/@status = "expired"
       return [ u: $u ])
          delete { $dot#u/file_list }
```

Figure 5-4. Core expression for normalized FLWUpdate.

The normalized core expression for the FLWUpdate is shown in Figure 5-4. The outer “for-tuple” loop iterates over the sequence of tuples of generated by the inner “for” loop. The “[u: $u]” represents the tuple sequence. For each iteration, the “$dot” variable binds to the current tuple, and “#u” extracts its corresponding value from the tuple sequence. Once the update AST is normalized, it is then ready for execution.

Execution

In the final update processing phase, the evaluation engine executes the normalized AST to modify the data model instance in memory. The actual modifications to underlying data model are carried out by *data model primitives*. The data model primitives comprise of IUD (insert/update/delete) functions. We specify the API for update primitives that must be supported by the underlying XML data store. In this prototype, we have implemented the primitives using tree-based, recursive algorithms. The evaluation engine can access these primitives and invoke them while processing the updates.

The data model primitives and the evaluation algorithms have been explained in greater detail in the following sections.

Data Model Primitives

Runtime updates to the underlying XML data model instance are performed by the data model primitives. These primitives support basic data modification operations such

as insert, delete, and replace. These operations must be executed atomically. The data model primitives are available in the form of library functions in Galax. The API for update primitives is outlined in Table 5-1.

Table 5-1. Update API for data model operations

Operation	Primitive
Insert	insert (ItemSequence update-content, Node parent-node, Node sibling-node)
Delete	delete (Node target-node)
Replace Node	replace_node (ItemSequence update-content, Node target-node, Node parent)
Replace Value	replace_value (ItemSequence update-content, Node target-node)

The signatures of the functions in the data model Update API contain "Node" and "ItemSequence" types, which are part of the XQuery data model. The Node type corresponds to the seven XQuery node types, viz. *document, element, attribute, namespace, processing instruction, comment,* and *text*. Update-content consists of a sequence of items, where an item can be of type *Node* or an *Atomic Value*[5]. The values of the target, parent, and/or sibling nodes are computed by the evaluation engine and passed as parameters to an update primitive. The update primitive performs the necessary transformations on the data model instance (which is represented using a XML tree structure in memory) and returns control to the evaluation engine for further processing.

Every XML document is represented in the XQuery data model by a tree structure that portrays its element hierarchy. We explain the algorithms the insert, delete, and replace primitive algorithms using examples based on the XML document instance and its tree representation shown in Figure 5-5. The document has a root node with tag "a" and elements with tag "b" and "f" are its children nodes. Elements "c", "h", and "i" are descendants having attribute nodes "d" and "k", while nodes with tags "e" and "k" are text nodes.

[5] Atomic Value corresponds to SimpleType in XML Schema.

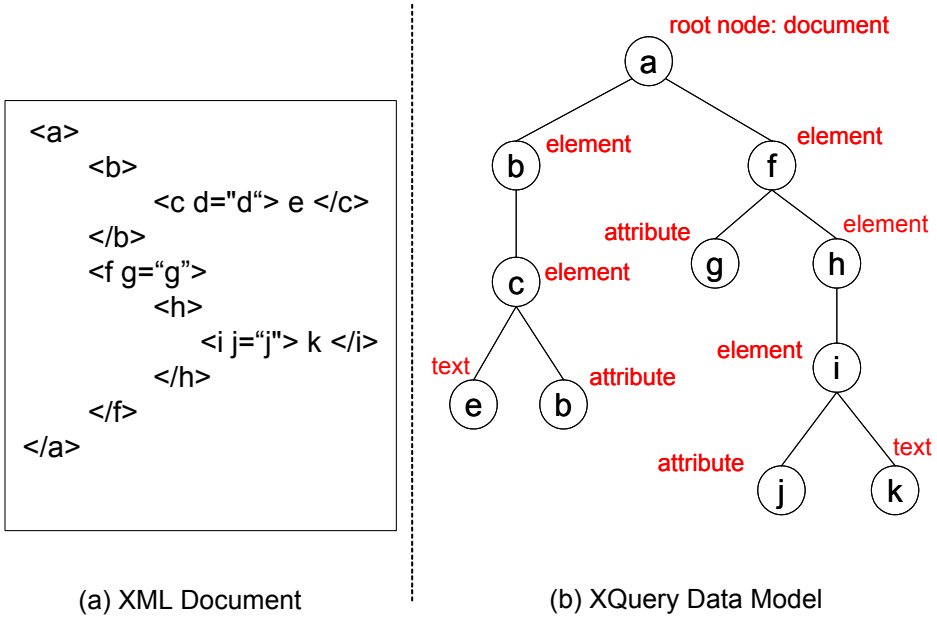

Figure 5-5. XQuery data model representation of an XML document.

Insert Primitive

The insert primitive adds new content to an XML tree. Its signature is given below:

- `insert (ItemSequence update-content, Node parent-node, Node sibling-node)`

The *update-content* consists of a sequence of items, which are inserted as children of the parent node. The insert operation is deterministic. It preserves the order of children nodes in the parent node. The insert algorithm is as follows:

1. Create a deep copy of the sequence of items to be inserted, viz. copy of update-content.
2. Check the node kind of the parent; the parent node should be a document or an element node, otherwise raise an error.
3. Compute child axis of the parent node, to retrieve its children.
4. If sibling node is specified, determine its position in the children node sequence.
5. Compute the insert location as follows: if the sibling node parameter is null, then the insert location is after the last child of the parent node, other wise the insert location is before the current position of the sibling node in document order.

6. Insert the items in the update-content recursively at the insert location.

7. Update the parent pointers of the newly inserted items.

It is important to note that the syntax for simple insert allows insertion into, as last, as first, before or after the target node. Although this may indicate the need for multiple insert primitives, we have implemented a single primitive that simulates all the required variations. The evaluation engine intelligently determines the values of the insert location, parent node, and target nodes which are passed to the insert primitive when it is invoked during update execution. For example, consider the following insert statement:

```
for $x in $doc
  insert <p>
           <q/>
           <r/>
         </p>
  after $x/d
```

The tree modification process for the above insert statement is shown in Figure 5-6.

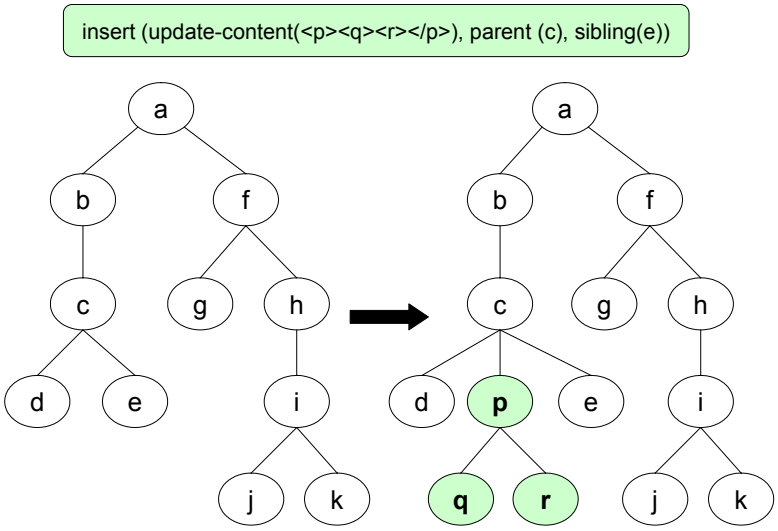

Figure 5-6. Tree modification by insert primitive.

When the insert is executed on the document in Figure 5-5, the evaluation engine determines that the parent node is element “c”, the insert location is a child of c, and the

value of sibling node is “e”. The ***after*** clause is simulated with the ***before*** clause using the insert primitive: `insert(update-content(<p><q/><r/></p>),parent(c),sibling(e))`.

Delete primitive

The delete primitive removes an existing XML node from the document tree. Its signature is as follows:

- `delete (Node target-node)`

The *target-node* in the delete primitive is passed as a parameter in the function call. If the target-node of a delete command has sub-elements, then the delete primitive deletes its children prior to deletion of the target-node itself. The algorithm for delete is as follows:

1. Check if the target-node is a valid, i.e. has type “Node”. If it has type “Atomic Value” then an error is raised.
2. Recursively descend down the sub-tree rooted at the target node, and mark each node as "deleted" (tail recursion).
3. Determine the parent of target node.
4. If parent exists, perform a clean up pass over the sub-tree rooted at the parent which removes the deleted child (target node) from the child axis of the parent. Maintain the relative order of the sub-elements. Otherwise the entire XML fragment is deleted.

For example, consider the following simple delete statement:

```
delete $doc/f/h
```

When the delete is executed on the document in Figure 5-5, the evaluation engine determines the location of node “h” in the `$doc/f` hierarchy and invokes the delete primitive: `delete(node(h))`. The delete primitive removes all the descendant nodes (i.e. nodes with tags ‘i’, ‘j’, and ‘k’) and the target-node (i.e. node with tag ‘h’) from the node hierarchy. The modified document tree is shown in Figure 5-7.

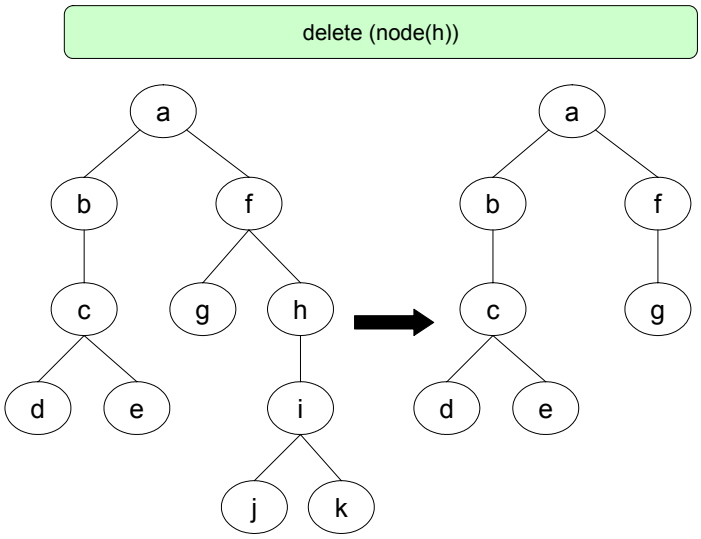

Figure 5-7. Tree modification by delete primitive.

Replace primitive

A replace operation can have two distinct semantics: (1) replace a node in the tree, (2) replace the value of a node in the tree. Hence two primitives are available to execute a replace function call. The evaluation engine invokes an appropriate primitive depending on occurrence of the "value of" construct in the replace AST. The signatures of the replace primitives are as follows:

- `Replace_node(ItemSequence update-content, Node target-node, Node parent-node)`
- `Replace-value(ItemSequence update-content, Node target-node)`

The "replace node" algorithm locates the target node in the child axis of the parent and substitutes it with the replace content, and finally updates the parent pointers of the new nodes. The "replace value" substitutes the child axis of the target node with the new content and updates their parent pointers.

For example, consider the following replace statement:

```
replace value of $doc//g with <m/><n/>
```

When the replace is executed on the document in Figure 5-5, the evaluation engine evaluates the `$doc//g` XQuery expression to determine the target node and invokes the primitive: `replace-value(update-content(<m/><n/>),node(g))`. The modified document tree is shown in Figure 5-8.

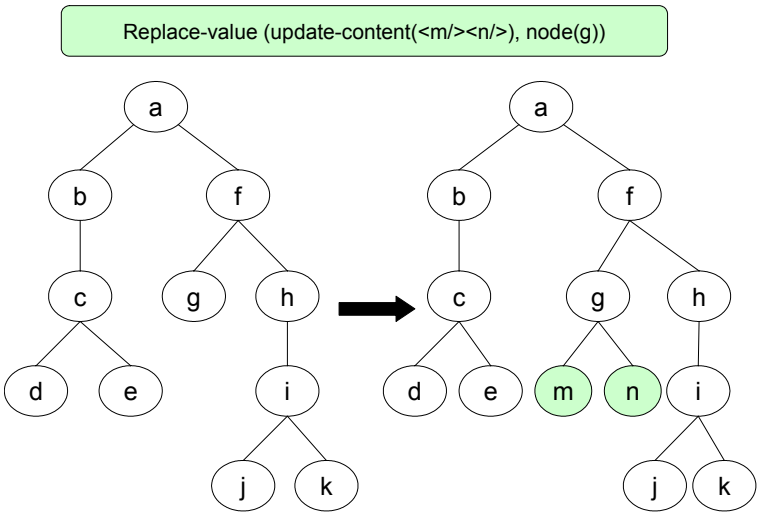

Figure 5-8. Tree modification by replace-value primitive.

We can leverage the basic algorithms for insert, delete and replace to implement the other update operations such as move and replace.

- **Move**: The move operation moves an existing sub-tree from one location to one (or multiple) new location(s). This can be simulated by an insertion(s) of a copy of the sub-tree to a new location followed by a deletion of the original sub-tree.
- **Rename**: The rename operation replaces the qualified name of an XML element. It can be simulated by creating a copy of the selected sub-tree, replacing the tag of the element or attribute node in the copy, then deleting the original sub-tree, and inserting the modified copy at the same location.

Having covered the basic algorithms for insert, delete, and replace primitives, we describe the update execution process. The input to the evaluation engine consists of a normalized update AST and XQuery data model. The evaluation engine executes the update AST by invoking appropriate primitives to modify the data model instance.

Update Execution

After an input update statement is parsed and normalized, it is executed by the evaluation engine. Execution of an update statement consists of the following three logical steps:

- Pre-update processing
- Semantic checking
- Update evaluation

Pre-update Processing

In the pre-update processing step, the XQuery Core expressions (target nodes and/or update content) in the normalized AST representation of the update statement are evaluated.

Semantic Checking

In the second step, semantics checks are applied to ascertain the validity of the target nodes and the update content. The semantic requirements for each type of update have been described in the previous chapter. The evaluation engine performs all the necessary checks and if any requirement is not met, then the update execution is aborted and an update error is raised.

Update Evaluation

In the final step, the updates are executed by applying modifications to the data model instance in memory. The evaluation engine makes calls to the data model primitives by invoking the appropriate function based on the "type" of the update AST. The target nodes and update content are passed as parameters to the data model primitive. The invoked primitive executes the update by performing necessary modifications to the tree-based data structures representing the data model instance and the result of the update is made persistent.

After the update operation is completed, the update processor may choose to re-evaluate node-identities in the data model, as every node in the XQuery data model must have a unique identity value. The node-identity re-evaluation can be postponed if the processor chooses to execute batched or complex updates.

The complete algorithms for evaluation of simple and complex updates are described in the following sections.

Algorithm for Evaluation of Simple Updates

The basic algorithm for the evaluation of Simple Updates is as follows:

5. Determine "type" of simple update.
6. If type is "insert" then do the following:
    a. Evaluate core expression for update-content.
    b. Evaluate core expression for insert-location.
    c. Check insert-location value, if it is null or empty raise an error.
    d. If insert-location corresponds to "into" or "as last into" clause, then the retrieve the parent-node from the insert location and check the node kind of the parent node. If the parent-node is a document or element node, then invoke the insert data model primitive, viz. insert(update-content, parent-node, null).
    e. If insert-location corresponds to "as first into" clause, then retrieve the parent-node from the insert location and compute its child axis. The first child in the node list becomes the sibling node of the insert and update-content is inserted by invoking the insert primitive, viz. insert(update-content, parent-node, sibling-node).
    f. If the insert-location corresponds to "before" clause, check if its type is an element, comment or processing-instruction node-type. If so re-compute the insert location by retrieving the parent node and invoking the insert primitive, viz. insert(update-content, parent node, before-node).
    g. If the insert-location corresponds to "after" clause, check if its type is an element, comment or processing-instruction node-type. If so re-compute the insert location by retrieving the parent-node and the preceding sibling of the after-node; invoke the insert primitive, viz. insert(update-content., parent-node, before-node).

7. If the type is “delete”, then do the following:

    a. Evaluate the core expression for delete target list.

    b. Check whether each target node is a valid node.

    c. For each node in the delete list, invoke the delete primitive, viz. delete(target-node).

8. If the type is “replace”, then do the following:

    a. Compute core expression for update-content.

    b. Compute the target-node of replace.

    c. Determine the “type” of replace. If it corresponds to “replace value of” clause, then check if the target-node is an element or attribute node and invoke the replace primitive, viz. replace-value(update-content, target-node), else check if the target-node is a document, element, or attribute node and invoke the replace primitive, viz. replace-node(update-content, target-node).

Algorithm for Evaluation of FLWUpdates

The algorithm for FLWUpdate execution takes the snapshot semantics into account, as the scope of the update is bound by the variable bindings in the FLWUpdate statement. Consider the generic FLWUpdate statement shown in Figure 5-9.

```
update
for $var1 in expr1
let $var2 := expr2
    (insert expr3 into | as last into | as first into | before | after expr4
     | delete expr5
     | replace <value of> expr6 with expr7)*
```

Figure 5-9. Generic FLWUpdate statement.

During the execution of the FLWUpdate, the variables in the *for* and *let* clauses variables are bound to values computed from an initial snapshot of the underlying data model (values are computed by evaluating query expressions) and then updates are processed. The execution algorithm consists of three logical steps: pre-update processing, semantic checking, and updates evaluations, which are shown in Figure 5-10.

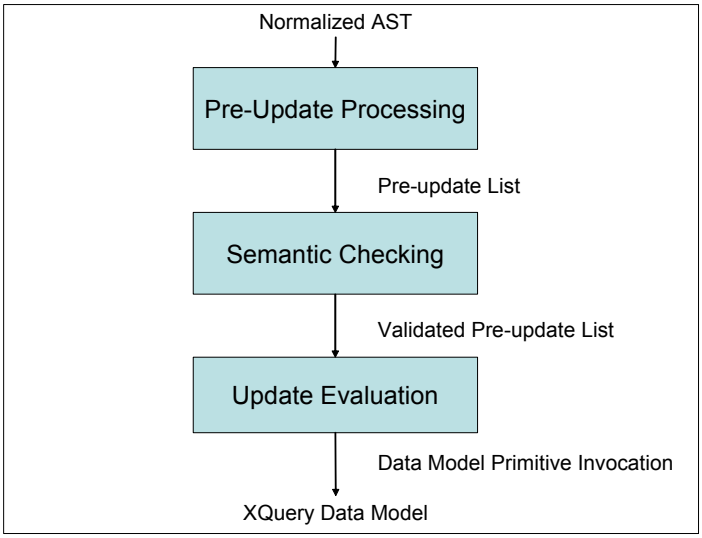

Figure 5-10. Update execution of FLWUpdate.

Pre-update Processing of FLWUpdate

In the pre-update processing step, target nodes and new update content for each simple update in the update list are evaluated and stored in an internal data-structure named "pre-update list". The processing occurs in the following manner:

- The variable bindings in the for and let clauses of the update statement, which define the scope of the update, are established by evaluating the query expressions and stored in a tuple sequence. (e.g. $var1 and $var2 bindings)
- The evaluation engine iterates through the list of simple updates and evaluates the sub-expressions in each simple update statement, with respect to the values in the tuple sequence. (e.g. expr3, expr4, expr5, expr6 and expr7)
- The resulting values are stored in the pre-update list according to the order of the update statements in the simple update list.

Semantic Checking of FLWUpdate

In the second step, semantic checks are applied to assert the validity of the target nodes and the update content in each of the simple update statements in the update list. The evaluation engine performs all the necessary checks on the values in the pre-update list. If any requirement is not met, then the update execution is aborted and an update error is raised.

Update Evaluation of FLWUpdate

In the final step, the simple updates are evaluated sequentially in the order in which they are specified. The evaluation engine iterates through the simple update list and invokes the data model primitives for each tuple binding in the pre-update list.

The pre-update list used in the algorithm holds the values of the query expressions based on the initial snapshot, before the evaluation engine begins execution of the FLWUpdate statement. Thus, the algorithm is able to enforce the snapshot rules for semantic consistency.

The distinction made between low-level data model operations and a higher-level update evaluation provides a degree of independence with respect to the physical implementation of the update processor. The data model primitives can be implemented using in-memory, tree-based algorithms (this prototype), on top of indexed XML structures in a native XML repository, or by converting them into SQL DML queries which can be issued to a relational database system. For example, the delete primitive can be implemented as a pre-delete SQL trigger that can be fired when parent-child nodes of an XML document are stored in the form of tuples in a relational database [14] and a parent tuple is deleted. The body of the trigger contains SQL code to delete the relevant tuples from the child relations. Moreover, since data model operations execute atomically, it becomes possible to define transactions over the XML data model by enhancements to the update processing model.

CHAPTER 6
EVALUATION OF UPDATE PROTOTYPE

In order to determine the accuracy of the update algorithms, we conducted comprehensive tests. We evaluated the update implementation based on two metrics: *correctness* and *completeness*. We define the execution of updates to be *correct*, if the outcome of the execution matches expected results; and we define our update language implementation to be *complete*, if it adheres to the update language specification. To verify the correctness of update execution, we performed experiments using XML document instances generated from UbiData profile repository. The tests were conducted on XML documents that fit into memory. We also conducted additional tests based on the recently announced XML Benchmark, also known as XMark [31]. The benchmark is designed top-down taking standardization issues around XML as a starting point and has been widely accepted in the research community. XMark emphasizes that the missing support for updates in current W3C XML query specification is a prominent one. We demonstrate (using update queries for XMark) that the update language implemented in this thesis makes the XQuery language complete.

Correctness

We used the UbiData user profiles as XML document instances to test the validity of the prototype implementation. The profile schema was incorporated into the system. We generated update queries based on profile use case scenarios. The update results were matched with results of our current user profile application, which was developed using a native XML repository named Xindice and XML C library API (libxml).

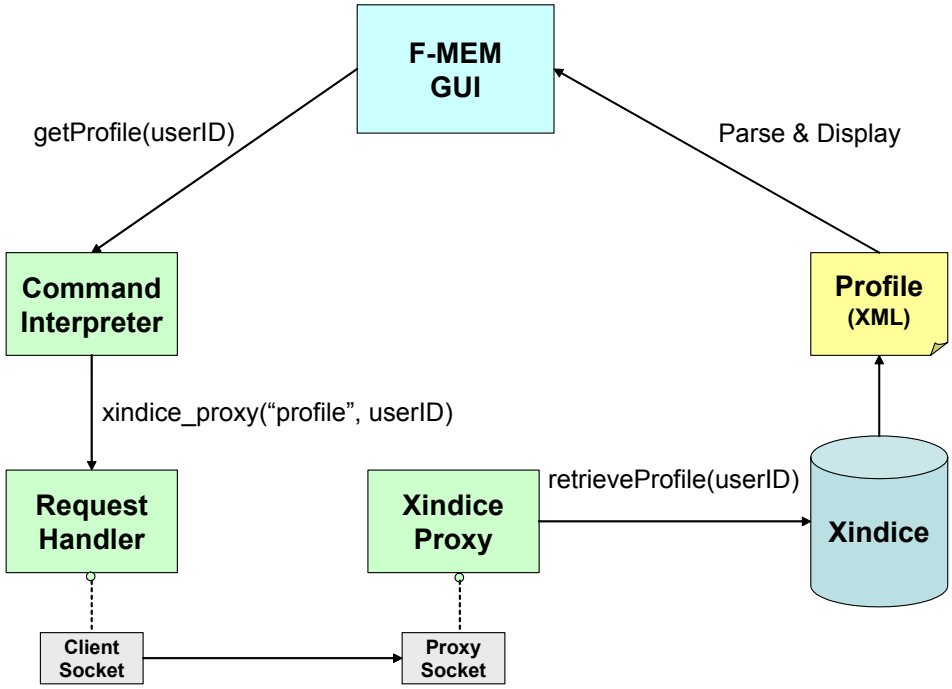

Figure 6-1. Profile management using Xindice native XML repository.

Before describing our experiments, we briefly outline the functionality of the testbed (Figure 6-1). The user profiles are stored in Xindice and they are queried, modified, and displayed using the F-MEM graphical user interface. When any command is issued by the F-MEM GUI (e.g., add new device and change user preferences), it is received by the command interpreter and translated into an appropriate request by the request handler. Since the XML library API modules are written in C, we use a proxy to communicate with Xindice, as the Xindice API is written in Java. When the intermediate proxy receives a profile retrieval request, it forwards the request to Xindice. The user profile is retrieved using an XPath query in the form of an XML document, the changes are applied and the modified document is saved in the repository. The saved results can be viewed using the F-MEM user interface.

In our experiments, we conducted a feature-wise examination of the update constructs to determine if all the simple and complex updates worked correctly. We invoked the data model primitive functions using various simple update statements. We checked the five insert clauses (viz. insert into, as first into, as last into, before, and after), the delete statement, and replace node/value functions for correct semantic execution. Then the complex updates were checked using different combinations of simple update statements. The conditional update tests included checks for effective boolean value expressions, mutually exclusive execution of then and else constructs, etc., where as the FLWUpdates tests included checks for correct execution order of simple updates, conformance to snapshot semantics, node tests, etc. Also various update processing modules, like parsing, mapping and evaluation were checked for functional correctness. The various use case scenarios used to generate update queries are outlined in Table 6-1.

Table 6-1. Profile use cases for update queries

	Use Case	UP Update Scenario
1	User Registration	Add a new UP to the profile repository, which includes the user and device information
2	UWS File Changes	Add/remove a file descriptor element to/from the UP every time a user creates/deletes a file
3	Incremental Updates	Update file descriptor parameter values, like file/replica version, hybrid priority value, last modification time, etc. in UP
4	Device List Changes	Insert, delete or update device descriptors in the UP when devices are added, removed or changed by the user
5	Synchronization	Update the file status, version, etc. parameters when sync operations are initiated or the UWS is updated
6	Asserting User Preferences	Update UP whenever user changes her preferences, like, update interval, password, etc.
7	File Sharing	When a shared file is updated, update multiple UP based on file and group ownership or permissions
8	Bookkeeping Operations	Delete “expired” UPs from the repository

The update queries for these use cases are described as update language examples in Chapters 4 and 5. We executed all the user profile update queries and compared the results with the expected outcome obtained using application code. Improvements to the

prototype were carried out and testing was conducted till the implementation of various features of the language was verified and the results were considered satisfactory.

Completeness

In addition to the UbiData profile updates, we conducted tests using the XML Benchmark (XMark), to evaluate the completeness of the update language. XMark provides a broad spectrum of queries, based on real world application scenarios, that challenges a major component of an XML database, like an XML storage engine or a query processor. Since XMark is primarily a benchmark for XML "queries", we added our own *update queries* to the framework to challenge the update processor on the expressiveness of the update language.

The benchmark document is modeled after a database deployed by an Internet auction site. The main entities are: item, open auction, closed auction, person and category. *Items* are objects that are on sale or that have been already sold, and each carries a unique identifier and bears properties like payment, credit card, money order, etc. *Open auctions* are auctions in progress. Their properties are the privacy status, bid history, along with references to the bidders and sellers, the current bid and default increase, the type of auction, status of transaction, and a reference to the item being sold. *Closed auctions* are auctions that are finished. Their properties are the seller and buyer references, sold item reference, date when the transaction was closed, its type, and the annotations after the bidding process. *Persons* are characterized by personal details, credit card number, profile of their interests, and a set of open auctions they watch. *Categories* feature name and descriptions used to classify items. A *category-graph* links categories into a network. The hierarchical schema is depicted in Figure 6-2.

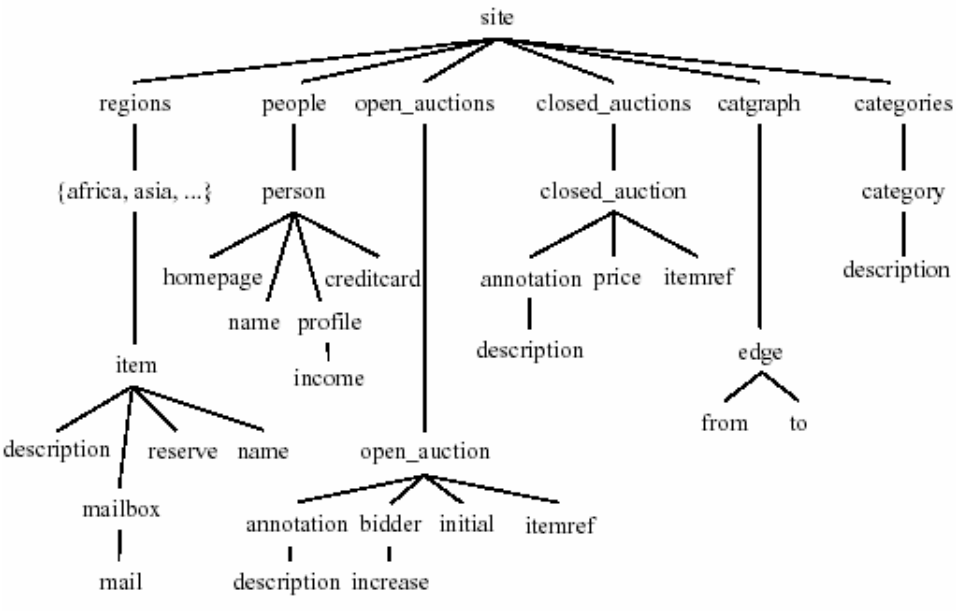

Figure 6-2. Hierarchical schema of entities in the XMark database.

All the entities and their relationships in the XMark document are expressed using XQuery type definitions in the update queries. As the XMark tests include only *benchmark queries using XQuery*, we executed a workload consisting of *update queries* on our update processor. Some of the update statements are listed below.

S1. Basic update: Insert a new item into the item database

```
insert
  <item id="item108">
    <location>brazil</location>
    <quantity>200</quantity>
    <name>XML in a Nutshell</name>
    <payment>Credit Card, Personal check</payment>
    <shipping>Will ship internationally</shipping>
    <incategory category="category1"/>
    <mailbox></mailbox>
  </item>
after $auction/site/regions/samerica/:item[@id="item107"]
```

S2. Comparison and Exact Match: If category "category4" exists, change its name to "2003 Car Sales" other wise insert a new category in the category list.

```
update
  if ($auction/site/categories/category[@id="category4"])
```

```
  then
    replace $auction/site/categories/category[@id="category4"]/name with
      <name>2003 Car Sales</name>
  else
    insert <category id="category4">
      <name>smoke awl convoy</name>
        <description>
          <parlist>
            <listitem>
          <text>cam trial <bold> dame </bold> thousands jewry</text>
        </listitem>
          </parlist>
        </description>
    </category> into $auction/site/categories
```

S3. Moving Sub-elements: Move an open auction to closed auction by selling the auction item to the last bidder

```
update
for $a in $auction/site
let $o := $a/open_auctions/open_auction[position()=2],
    $num := fn:count($a//closed_auction)
   insert <closed_auction>
        <auction_count>{$num + 1}</auction_count>
        <seller person="{$o/seller/@person}"></seller>
        <buyer person="{$o/bidder[last()]/personref/@person}"></buyer>
        <price>{$o/initial + $o/bidder[last()]/increase}</price>
        <annotation>Closed satisfactorily</annotation>
     </closed_auction> as last into $a/closed_auctions
   delete $o
```

S4. Complex Results: For each person, compute the total purchase amount and insert it into his/her profile.

```
update
  for $p in $auction/site/people/person
  let $s := fn:sum(for $o in $auction/site/closed_auctions/closed_auction
                where $o/buyer/@person=$p/@id
                return $o/price)
    insert <purchase_history>{ $s }</purchase_history> into $p
```

S5. **Joins:** For each region, compute total sales (join: closed_auction x regions)

```
update
  for $c in $auction/site/closed_auctions/closed_auction
  let $r := $auction/site/regions//:item[@id=$c/itemref/:item][total_sales]
    replace value of $r/total_sales with ($r/total_sales + $c/price);
update
  for $c in $auction/site/closed_auctions/closed_auction
  let $r :=$auction/site/regions//:item[@id=$c/itemref/:item][not(total_sales)]
    insert <total_sales>{$c/price}</total_sales> into $r
```

The XMark update queries have been classified into various categories based on the nature of the update. In the previous examples, update statements S1, S2, and S3 apply

modifications to single elements in the document where as S4 and S5 apply modifications across various parts of the document. The updates try to capture the essential primitives of XML update processing, such as single point updates, value and tag modification, sub-element reconstruction, maintaining sort order during insertion and replacement, sequential and consecutive updates in complex updates, etc. Also note that current syntax of the complex update construct allows only simple updates in its update list. In future, we hope that the syntax will allow *nested complex updates*, which are necessary while computing joins over various relationships in data-centric documents.

We conclude that the prototype implementation completely conforms to the update language specification. The update language used in the prototype is based on an initial W3C working draft on XML updates, which will be published in the near future. We have implemented all the constructs in the language and tested them for correctness. The update language can be used to sufficiently express updates on both 'data-centric' documents (i.e., documents which logically represent data structures and map very nicely to relational databases) and 'text-oriented' documents (i.e., natural language with mark-up interspersed). We were able to express all UbiData profile management queries and updates (which cater to both data-centric and text-oriented information) using the extended XQuery update language. For example, we directly express changes to the user, preferences, device and file elements of an individual profile using simple updates, while batched and incremental updates to profiles are grouped together using FLWUpdates.

CHAPTER 7
CONCLUSION AND FUTURE WORK

Conclusion

In this thesis, we have described a new approach to update processing for XML documents using a declarative update language based on XQuery 1.0. The language is easy to learn and powerful. It consists of various constructs, such as simple insert, delete, replace, conditional, and FLWUpdate, which can be used to efficiently modify XML data model instances. These constructs have been explained with the help of use case scenarios based on the UbiData mobile user profile application. The update prototype has been completely implemented on top of the Galax XQuery 1.0 query processor. The update processor is fully functional and the source code is available online at the Galax website[6].

One of the most important and lasting contributions of the update processing framework is that it eliminates tedious and ad hoc programming techniques (e.g. making updates to XML using DOM primitives) that are currently used to update XML documents. In addition, application developers benefit from use of an integrated query and update language to perform many different types of retrievals and modifications on XML documents (e.g. embedded XQuery queries or statements) in their software applications. Moreover, in this research, we have developed a vendor-independent, standard API to update native and relational XML repositories, which is beneficial to the

[6] http://db.bell-labs.com/galax

database community at large. Lastly, for researchers, we provide a flexible architecture to study the impact of updates on XML processing and scope for refinement or improvement of update processing algorithms.

Future Work

In addition to the various features that our update processor provides, we see several important ways our existing prototype implementation can be enhanced. We will briefly summarize our ideas in the following subsections.

Storage Manager for the Query Processor

Although the integrated query and update processor is completely functional, all XML processing functions are currently carried out only on XML documents that fit into the main memory. To fully study the impact of update processing on XML documents, the extended XQuery processor should be integrated with a storage manager and updates should be carried out on XML data contained with the storage component. We are currently using the Berkeley DB [32] commercially-supported embedded database to develop a XML storage manager for Galax. An OCaml wrapper has been developed to integrate Galax with Berkeley DB. XML documents will be stored in Berkeley DB using record descriptors, and queried as well as updated using B+ tree indexes.

Schema Validation

If the input XML document instances to the update processor conform to a specific XML schema, then the target nodes and value content of the update must be validated against the schema before the update execution is carried out. The current update prototype implementation lacks support for XML schema validation of update queries. Hence, there is scope for further research in the area of static and run-time schema validation during update processing.

Transaction Processing for XML Data

Concurrent query and update to XML data can be supported if efficient transaction processing models are developed to deal with semi-structured data. Our update processing framework is the first step in that direction. We have defined the basic, primitive update operations and storage-independent framework to support updates. Extensive research can be carried to develop efficient transaction support for the XQuery data model structures based on these primitives in the future.

Performance Analysis

In future, experiments can be conducted to analyze *performance* of the update processor. Various measures to critically evaluate various update processing algorithms and approaches must be established, viz. execution time, path traversals, node identity changes, etc. Then tests can be conducted using both *bulk* and *random* workloads. In a bulk workload, an update operation is applied to range of sub-tree elements in XML documents. For example, a 100% bulk delete would remove all elements in the tree except the root node. For random workloads, updates are carried out on randomly chosen sub-trees. XMark update queries can be used to complement the performance analysis and provide baseline performance figures.

APPENDIX A
DTD FOR UBIDATA USER PROFILE

```
<?xml version='1.0' encoding='UTF-8' ?>
<!ELEMENT user_profiles (user_profile)*>
<!ELEMENT user_profile (system_info? , user_info , device_list , file_list?)>
<!ATTLIST user_profile  status CDATA  #IMPLIED
                        userID CDATA  #REQUIRED >
<!ELEMENT system_info (update_frequency?)>
<!ELEMENT update_frequency (#PCDATA)>
<!ELEMENT user_info (username , address* , email* , phone* , password)>
<!ELEMENT username (first , last)>
<!ELEMENT first (#PCDATA)>
<!ELEMENT last (#PCDATA)>
<!ELEMENT address (addressline1 , addressline2? , city , state , zipcode)>
<!ELEMENT addressline1 (#PCDATA)>
<!ELEMENT addressline2 (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT zipcode (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT password (#PCDATA)>
<!ELEMENT device_list (device+)>
<!ELEMENT device (device_name , device_description? , device_capabilities)>
<!ATTLIST device  deviceID CDATA  #REQUIRED >
<!ELEMENT device_name (#PCDATA)>
<!ELEMENT device_description (#PCDATA)>
<!ELEMENT device_capabilities (cpu? , ram? , hdd? , os?)*>
<!ELEMENT cpu (#PCDATA)>
<!ELEMENT ram (#PCDATA)>
<!ELEMENT hdd (#PCDATA)>
<!ELEMENT os (#PCDATA)>
<!ELEMENT file_list (file+)>
<!ELEMENT file (creation_time , file_version , file_description? , file_status? ,
hybrid_priority , replica_list)>
<!ATTLIST file  fileID      CDATA  #REQUIRED
              groupID     CDATA  #IMPLIED
              owner       CDATA  #REQUIRED
              permissions CDATA  #IMPLIED >
<!ELEMENT creation_time (#PCDATA)>
<!ELEMENT file_version (#PCDATA)>
```

```
<!ELEMENT file_status (#PCDATA)>
<!ELEMENT hybrid_priority (recency? , frequency , active_period?)>
<!ATTLIST hybrid_priority  value CDATA  #REQUIRED >
<!ELEMENT recency (#PCDATA)>
<!ELEMENT frequency (#PCDATA)>
<!ELEMENT active_period (#PCDATA)>
<!ELEMENT replica_list (replica*)>
<!ELEMENT replica (location , last_modification_time , replica_state? ,
replica_version)>
<!ELEMENT location (#PCDATA)>
<!ELEMENT last_modification_time (#PCDATA)>
<!ELEMENT replica_state (#PCDATA)>
<!ELEMENT replica_version (#PCDATA)>
<!ELEMENT file_description (#PCDATA)>
```

APPENDIX B
XML SCHEMA FOR UBIDATA USER PROFILE

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema">
   <xsd:element name = "user_profiles">
      <xsd:complexType>
         <xsd:sequence minOccurs = "0" maxOccurs = "unbounded">
            <xsd:element ref = "user_profile"/>
         </xsd:sequence>
      </xsd:complexType>
   </xsd:element>
   <xsd:element name = "user_profile">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element ref = "system_info" minOccurs = "0"/>
            <xsd:element ref = "user_info"/>
            <xsd:element ref = "device_list"/>
            <xsd:element ref = "file_list" minOccurs = "0"/>
         </xsd:sequence>
         <xsd:attribute name = "status" type = "xsd:string"/>
         <xsd:attribute name = "userID" type = "xsd:string"/>
      </xsd:complexType>
   </xsd:element>
   <xsd:element name = "system_info">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element ref = "update_frequency" minOccurs = "0"/>
         </xsd:sequence>
      </xsd:complexType>
   </xsd:element>
   <xsd:element name = "update_frequency" type = "xsd:decimal"/>
   <xsd:element name = "user_info">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element ref = "username"/>
            <xsd:element ref = "address" minOccurs = "0" maxOccurs = "unbounded"/>
            <xsd:element ref = "email" minOccurs = "0" maxOccurs = "unbounded"/>
            <xsd:element ref = "phone" minOccurs = "0" maxOccurs = "unbounded"/>
            <xsd:element ref = "password"/>
         </xsd:sequence>
      </xsd:complexType>
```

```
</xsd:element>
<xsd:element name = "username">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element ref = "first"/>
         <xsd:element ref = "last"/>
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>
<xsd:element name = "first" type = "xsd:string"/>
<xsd:element name = "last" type = "xsd:string"/>
<xsd:element name = "address">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element ref = "addressline1"/>
         <xsd:element ref = "addressline2" minOccurs = "0"/>
         <xsd:element ref = "city"/>
         <xsd:element ref = "state"/>
         <xsd:element ref = "zipcode"/>
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>
<xsd:element name = "addressline1" type = "xsd:string"/>
<xsd:element name = "addressline2" type = "xsd:string"/>
<xsd:element name = "city" type = "xsd:string"/>
<xsd:element name = "state" type = "xsd:string"/>
<xsd:element name = "zipcode" type = "xsd:integer"/>
<xsd:element name = "email" type = "xsd:string"/>
<xsd:element name = "phone" type = "xsd:string"/>
<xsd:element name = "password" type = "xsd:string"/>
<xsd:element name = "device_list">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element ref = "device" maxOccurs = "unbounded"/>
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>
<xsd:element name = "device">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element ref = "device_name"/>
         <xsd:element ref = "device_description" minOccurs = "0"/>
         <xsd:element ref = "device_capabilities"/>
      </xsd:sequence>
      <xsd:attribute name = "id" type = "xsd:string"/>
   </xsd:complexType>
```

```
</xsd:element>
<xsd:element name = "device_name" type = "xsd:string"/>
<xsd:element name = "device_description" type = "xsd:string"/>
<xsd:element name = "device_capabilities">
   <xsd:complexType mixed = "true">
      <xsd:sequence>
         <xsd:element name = "cpu" type = "xsd:string" minOccurs = "0"/>
         <xsd:element name = "hdd" type = "xsd:string" minOccurs = "0"/>
         <xsd:element name = "ram" type = "xsd:string" minOccurs = "0"/>
         <xsd:element name = "os" type = "xsd:string" minOccurs = "0"/>
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>
<xsd:element name = "file_list">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element ref = "file" maxOccurs = "unbounded"/>
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>
<xsd:element name = "file">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element ref = "creation_time"/>
         <xsd:element ref = "file_version"/>
         <xsd:element ref = "file_description" minOccurs = "0"/>
         <xsd:element ref = "file_status" minOccurs = "0"/>
         <xsd:element ref = "hybrid_priority"/>
         <xsd:element ref = "replica_list"/>
      </xsd:sequence>
      <xsd:attribute name = "fileID" type = "xsd:string"/>
      <xsd:attribute name = "groupID" type = "xsd:string"/>
      <xsd:attribute name = "owner" type = "xsd:string"/>
      <xsd:attribute name = "permissions" type = "xsd:string"/>
   </xsd:complexType>
</xsd:element>
<xsd:element name = "creation_time" type = "xsd:string"/>
<xsd:element name = "file_version" type = "xsd:decimal"/>
<xsd:element name = "file_description" type = "xsd:string"/>
<xsd:element name = "file_status" type = "xsd:string"/>
<xsd:element name = "hybrid_priority">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element ref = "recency" minOccurs = "0"/>
         <xsd:element ref = "frequency"/>
         <xsd:element ref = "active_period" minOccurs = "0"/>
```

```
            </xsd:sequence>
            <xsd:attribute name = "value" type = "xsd:string"/>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name = "recency" type = "xsd:string"/>
    <xsd:element name = "frequency" type = "xsd:integer"/>
    <xsd:element name = "active_period" type = "xsd:string"/>
    <xsd:element name = "replica_list">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref = "replica" minOccurs = "0" maxOccurs = "unbounded"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name = "replica">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref = "location"/>
                <xsd:element ref = "last_modification_time"/>
                <xsd:element ref = "replica_state" minOccurs = "0"/>
                <xsd:element ref = "replica_version"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name = "location" type = "xsd:string"/>
    <xsd:element name = "last_modification_time" type = "xsd:string"/>
    <xsd:element name = "replica_state" type = "xsd:string"/>
    <xsd:element name = "replica_version" type = "xsd:decimal"/>
</xsd:schema>
```

APPENDIX C
SAMPLE UBIDATA USER PROFILE

```
<?xml version = "1.0" encoding = "UTF-8"?>
<user_profiles>
    <user_profile userID = "gsur" status = "active">
        <system_info>
            <update_frequency>500</update_frequency>
        </system_info>
        <user_info>
            <username>
                <first>Gargi</first><last>Sur</last>
            </username>
            <address>
                <addressline1>1111 SW 16th Avenue</addressline1>
                <city>Gainesville</city>
                <state>FL</state>
                <zipcode>32601</zipcode>
            </address>
            <email>gsur@cise.ufl.edu</email>
            <phone>352-392-6833</phone>
            <password>ER234#sft</password>
        </user_info>
        <device_list>
            <device deviceID = "laptop">
                <device_name>Dell Inspiron 4100</device_name>
                <device_capabilities>3Com 10 Mbps Ethernet</device_capabilities>
            </device>
            <device deviceID = "PDA">
                <device_name>Compaq iPAQ 3890</device_name>
                <device_capabilities>512Mb RAM</device_capabilities>
            </device>
        </device_list>
        <file_list>
            <file fileID="mfs://MDS/gsur/docs/abc.txt" owner = "gsur" permissions =
"rwxr-xr-x">
                <creation_time>Fri Oct 15 09:06:34 2002</creation_time>
                <file_version>1.1</file_version>
                <file_status>current</file_status>
                <hybrid_priority value = "1">
                    <frequency>10</frequency>
                </hybrid_priority>
```

```
            <replica_list>
               <replica>
                  <location>mfs://gsur/laptop/c:/abc.txt</location>
                  <last_modification_time>Fri Oct 25 10:35:23
2002</last_modification_time>
                  <replica_version>1.1</replica_version>
               </replica>
               <replica>
                  <location>mfs://gsur/PDA/delta/abc.txt</location>
                  <last_modification_time>Thurs Oct 24 11:36:23
2002</last_modification_time>
                  <replica_version>1.0</replica_version>
               </replica>
            </replica_list>
         </file>
         <file fileID="mfs://MDS/gsur/docs/arch.doc" owner="gsur" permissions="rwxr-
xr-x">
            <creation_time>Fri Oct 15 09:06:34 2002</creation_time>
            <file_version>1.0</file_version>
            <hybrid_priority value = "1">
               <frequency>15</frequency>
            </hybrid_priority>
            <replica_list>
               <replica>
                  <location>mfs://gsur/laptop/c:/arch.doc</location>
                  <last_modification_time>Wed Jun 25 10:35:23
2003</last_modification_time>
                  <replica_version>1.0</replica_version>
               </replica>
            </replica_list>
         </file>
         <file fileID = "mfs://MDS/gsur/ubidata/usecases.doc" groupID = "ubidata"
owner = "gsur" permissions = "rwxrwxr-x">
            <creation_time>Fri Oct 25 09:06:34 2002</creation_time>
            <file_version>1.0</file_version>
            <file_status>current</file_status>
            <hybrid_priority value = "2">
               <recency>Sat Oct 26 14:32:43 2002</recency>
               <frequency>4</frequency>
            </hybrid_priority>
            <replica_list>
               <replica>
                  <location>mfs://gsur/laptop/d:/ubidata/usecases.doc</location>
                  <last_modification_time>Fri Oct 25 10:35:23
2002</last_modification_time>
                  <replica_version>1.0</replica_version>
```

```
            </replica>
            <replica>
              <location>mfs://jhammer/laptop/c:/ubidata/usecases.doc</location>
              <last_modification_time>Fri Oct 25 1:00:23
2002</last_modification_time>
              <replica_version>1.0</replica_version>
            </replica>
          </replica_list>
        </file>
      </file_list>
    </user_profile>
</user_profiles>
```

APPENDIX D
UPDATE GRAMMER IN EBNF

```
[1]   Update            ::=  SimpleUpdate | ComplexUpdate
[2]   SimpleUpdate      ::=  Insert | Delete | Replace | EmptyUpdate
[3]   ComplexUpdate     ::=  FLWUpdate | ConditionalUpdate
[4]   FLWUpdate         ::=  "update" (ForClause | LetClause)+ WhereClause?
                                 SimpleUpdate+
[5]   ConditionalUpdate ::=  "update" <"if" "("> ConditionExpr ")"
                                 "then" SimpleUpdate "else" SimpleUpdate
[6]   ConditionExpr     ::=  Expr
[7]   Insert            ::=  "insert" UpdateContent InsertLocation
[8]   InsertLocation    ::=  ((<"as" "last"> | <"as" "first">)? "into" TargetNode)
                                 | ("after" TargetNode)
                                 | ("before" TargetNode)
[9]   UpdateContent     ::=  Expr
[10]  TargetNode        ::=  Expr
[11]  Replace           ::=  "replace" <"value" "of">? TargetNode
                                 "with" UpdateContent
[12]  EmptyUpdate       ::=  "(" ")"
[13]  Delete            ::=  "delete" TargetNodes
[14]  TargetNodes       ::=  Expr
```

LIST OF REFERENCES

1. World Wide Web Consortium, "Extensible Markup Language (XML) 1.0 (Second Edition)." W3C Recommendation, 6 October 2000. Available at http://www.w3.org/TR/2000/REC-xml-20001006.

2. International Organization for Standardization, "Information Processing – Text and office systems – Standard Generalized Markup Language (SGML)." ISO 8879:1986, 154pp, 1986.

3. Software AG, "Tamino XML Server," http://www.softwareag.com/tamino.

4. Progress Software, "ObjectStore," http://www.objectstore.net.

5. S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon, "XQuery 1.0: An XML Query Language." W3C Working Draft, May 2003. Available at http://www.w3.org/TR/xquery.

6. A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Simeon, "XML Path Language (XPath) 2.0." W3C Working Draft, May 2003. Available at http://www.w3.org/TR/xpath20.

7. M. F. Fernandez, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh, "XQuery 1.0 and XPath 2.0 Data Model." W3C Working Draft, May 2003. Available at http://www.w3.org/TR/xpath-datamodel.

8. D. Draper, P. Fankhauser, M. F. Fernandez, A. Malhotra, K. Rose, M. Rys, J. Simeon, and P. Wadler, "XQuery 1.0 and XPath 2.0 Formal Semantics." W3C Working Draft, 2 May 2003. Available at http://www.w3.org/TR/xquery-semantics.

9. Galax, "Galax, An Implementation for XQuery," http://db.bell-labs.com/galax.

10. A. Helal, J. Hammer, J. Zhang, and A. Khusraj, "Three-tier Architecture for Ubiquitous Data Access." In *Proceedings of the First ACS/IEEE International Conference on Computer Systems and Applications*, Beirut, Lebanon, pages 177-180, June 2001.

11. XML:DB, "XUpdate - XML Update Language," http://www.xmldb.org/xupdate.

12. XML:DB, "XML:DB Initiative for XML Databases," http://www.xmldb.org.

13. I. Tatarinov, Z. Ives, A. Halevy, and D. Weld, "Updating XML." In *Proceedings of ACM SIGMOD Conference*, Santa Barbara, California, pages 413-424, May 2001.

14. J. Shanmugasundaram, H. Gang, K. Tufte, C. Zhang, D. J. DeWitt, and J. F. Naughton, "Relational Databases for Querying XML Documents: Limitations and Opportunities." In *Proceedings of 25th International Conference on Very Large Data Bases (VLDB'99)*, Morgan Kaufmann, pages 302-304, September 1999.

15. P. Lehti, "Design and Implementation of a Data Manipulation Processor for an XML Query Language." Technical Report, Report KOM-D-149, Technische Universitat Darmstadt, August 2001.

16. World Wide Web Consortium, "XML Syntax for XQuery 1.0 (XQueryX)." W3C Working Draft, June 2001. Available at http://www.w3.org/TR/xqueryx.

17. Software AG, "QuiP," http://developer.softwareag.com/tamino/quip.

18. M. Rys, "Proposal for an XML Data Modification Language." Microsoft Corp., Redmond, WA, Proposal May 2002.

19. Liberty Alliance Project, "Open, Interoperable Standards for Federated Network Identity," http://www.projectliberty.org.

20. 3rd Generation Partnership Project (3GPP), "3GPP Generic User Profile (GUP)." TS 22.240, stage 1, June 2003.

21. A. Sahuguet, R. Hull, D. Lieuwen, and M. Xiong, "Enter Once, Share Everywhere: User Profile Management in Converged Networks." In *First Biennial Conference on Innovative Data Systems Research (CIDR),* Asilomar, CA, January 2003.

22. M. Lanham, A. Kang, J. Hammer, A. Helal, and J. Wilson, "Format-Independent Change Detection and Propagation in Support of Mobile Computing." In *Proceedings of XVII Brazilian Symposium on Databases (SBBD)*, Gramado, Brazil, pages 27-41, October 2002.

23. J. Zhang, A. Helal, and J. Hammer, "UbiData: Ubiquitous Mobile File Service." In *Proceedings of ACM Symposium on Applied Computing (SAC)*, Melbourne, FL, pages 45-56, March 2003.

24. T. Imielinski and B. R. Badrinath, "Data Management for Mobile Computing." In *SIGMOD Record,* Vol. 22, no. 1, pp. 34-39, March 1993.

25. J. Jannink, D. Lam, N. Shivakumar, J. Widom, and D. Cox, "Data Management for User Profiles in Wireless Communication Systems." Technical report, Stanford University, Computer Science Dept., October 1995. Available at http://www-db.stanford.edu/pub/jannink/1995/profile.ps.

26. G. Sur and J. Hammer, "Management of User Profile Information in UbiData." Dept. of CISE, University of Florida, Technical Report TR03-001, January 2003. Available at ftp://ftp.cise.ufl.edu/cis/tech-reports/tr03/tr03-001.pdf.

27. J. Kistler and M. Satyanarayanan, "Disconnected Operation in the Coda File System." In *ACM Transactions on Computer Systems*, Vol. 10, No. 1, February 1992.

28. Apache XML Project, "Apache Xindice," http://xml.apache.org/xindice.

29. M. Fernandez, J. Simeon, B. Choi, A. Marian, and G. Sur, "Implementing XQuery 1.0: The Galax Experience." To be published in *Proceedings of International Conference on Very Large Data Bases (VLDB'03)*, Berlin, Germany, September 2003.

30. X. Leroy, "The Objective Caml system, release 3.06, Documentation and User's Manual." Institut National de Recherche en Informatique et en Automatique, August 2002.

31. A. Schmidt, F. Waas, M. Kersten, D. Florescu, I. Manolescu, M. Carey, and R. Busse, "XMark: A Benchmark for XML Data Management." In *Proceedings of International Conference on Very Large Data Bases (VLDB'02)*, Hong Kong, pages 974-985, August 2002.

32. M. Olson, K. Bostic, and M. Seltzer, "Berkeley DB." In *Proceedings of 1999 USENIX Annual Technical Conference*, Monterey, CA, pages 183-192, June 1999.

BIOGRAPHICAL SKETCH

Gargi Sur was born in Hooghly, West Bengal, India. She received her Bachelor's Degree in computer science with honors from the Maharaja Sayajirao University of Baroda, Gujarat, India in May 2001.

In August 2001, she joined the University of Florida to pursue a Master of Science degree in computer science. She has been a Research Assistant during her studies at the University of Florida. In the summer of 2002, she interned at Lucent Bell Laboratories. Her research interests include XML data processing, relational database systems, query processing and optimization, distributed systems, and mobile networks.