

Moving From XML Documents to XML Databases

by

Fengdong Du

B.Eng., Dalian University of Technology, China

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
Master of Science

in

THE FACULTY OF GRADUATE STUDIES
(Department of Computer Science)

We accept this thesis as conforming
to the required standard

The University of British Columbia

March 2004

© Fengdong Du, 2004

Abstract

XML has become a standard format in information exchange and integration. Database support of persistent data storage and query capability is often desired for many XML applications. While it is possible to store XML data in traditional relational databases or object-oriented databases, we also desire high-performance native XML databases that are particularly tailored for XML data.

We believe an ideal XML database needs to be generic, automatic, access transparent and performance transparent. With these features as the fundamental success criteria for XML databases, this thesis discusses various existing XML database solutions. With these features as design goals, we develop a native XML database that includes native XML storage and an implementation of the XQuery language. This native database is generic in the sense that it accepts any well-formed XML data, regardless of the actual structure of data, and does not rely on schema existence. This database system is automatic because it does not require any database physical design works. Retrieving XML documents or fragments is solely through the XQuery interface. Query performance is transparent with respect to the structure of data.

Contents

Abstract	ii
Contents	iii
List of Tables	vi
List of Figures	vii
1 Introduction	1
2 XML Data As Objects	6
2.1 XML Data as Objects	6
2.2 Document Object Model	7
2.3 DOM Persistence on Disk	9
2.4 SQL:1999, Object-Relational Model	11
2.4.1 User-defined Data Types	12
2.4.2 Object References	13
3 A Case Study: Oracle XML DB	14
3.1 Storage	15
3.2 Schema Mapping and Table Creation	16

3.3	Index and Query Evaluation	17
3.4	Conclusion	19
4	Native Works	21
4.0.1	Berkeley DB XML	22
4.0.2	eXist	22
4.0.3	Apache Xindice	23
5	XQuery and XPath Language	24
5.1	XQuery and XPath Data Model	24
5.1.1	The Atomic Items	25
5.1.2	Data Type and Naming	25
5.1.3	The Node Items	26
5.2	The XPath Language	27
5.3	The XQuery Language	28
6	This Native XML Database	29
6.1	System Architecture	29
6.1.1	Overview	29
6.1.2	The Data Storage Part	29
6.1.3	The Query Evaluation Part	31
6.2	Database Design	32
6.3	Genericness and Automation	34
6.4	Performance Measurement	35
6.5	Summary	36
	Bibliography	39

List of Tables

2.1	Composition Relationship of DOM Node Objects	8
5.1	Built-in Atomic Data Types in XQuery and XPath Data Model . . .	26
5.2	Seven XML Node Types in XQuery and XPath Data Model	27
6.1	Server-side API of This Native Database	30

List of Figures

6.1	System Architecture of This Native Database	37
6.2	Storage Size vs. Document Size	38
6.3	Time to Build Indices vs. Number of Objects	38
6.4	Query Time vs. Path Length and Number of Objects	38

Chapter 1

Introduction

XML has become a standard format in cross-platform information exchange and integration. Nowadays, many applications produce new data directly in XML format or convert existing information into XML. As a consequence, a critical problem arises for large amounts of data, where data persistence is desired. Namely, how do we store the XML data and manage it? Due to the variety of XML applications, an ideal solution needs to be generic, and provide an open interface for data manipulation.

Depending on the characteristics of XML applications, the current XML storage techniques can be classified into two major categories. Most text-centric applications (e.g., newspapers) choose an existing file system for data storage. Data is usually divided into logical units, and each logical unit is physically stored as a separate file. As an example, a newspaper application may divide the entire year's newspapers into 12 collections by months, and store each collection as a document file. This type of application usually provides a keyword-based search tool and manipulates the data in application-specific processes. While this approach simplifies the storage problem, it has some major drawbacks. First, storing XML data as plain text makes it difficult to develop a generic data manipulation interface.

Second, mapping logical units of data to individual files makes it difficult to view the data from a different perspective. For this reason, this type of application only provides services with limited functionalities and therefore restricts the usage of data.

On the other hand, in data-centric applications such as e-commerce applications, data is typically highly-structured, e.g., extracted from a relational database management system (RDBMS). XML is primarily used as a tool to publish data to the Web or deliver information in a self-descriptive way in place of the conventional relative files. This type of application relies on the RDBMS for data storage. Data received in XML format is eventually put into an RDBMS when persistence is desired. Over the years, an RDBMS has been well developed to efficiently store and retrieve well-structured data. Structured Query Language (SQL) and many useful extended RDBMS utilities (e.g., Programming Language SQL, stored procedures) act as an application-independent data manipulation interface. Applications can communicate with databases through this generic interface and, on top of it, provide services with very rich functionalities.

While storing XML data into an RDBMS can take advantage of the well-developed relational database techniques and open interfaces, this approach requires an extra schema-mapping process applied to XML data, which involves schema transformation and usually decomposition. The schemas of XML data have to be mapped to strictly-defined relational schemas before data is actually stored. This process is strongly application-dependent or domain-dependent because there must be enough information available to determine many relational database design issues such as which table in the target RDBMS is a good place to store the information delivered, what new tables need to be created, which elements/attributes should

be indexed, etc. No matter how this kind of information is obtained, whether delivered with XML data as schemas and processing instructions, or the application context makes it obvious, it is hard to develop an automatic and generic schema-mapping mechanism. Instead, application-specific work needs to take care of the schema-mapping problem. This involves non-trivial work of database server-side programming and database administration.

Another drawback of storing XML data in an RDBMS is that it is hard to efficiently support many types of queries that people want to ask on XML data. In RDBMS, each table has a pre-defined primary key field, and possibly a few other indexed fields. Queries not on the key field and not on the indexed fields will result in table scans (i.e., possibly a very large number of I/O's, which can be very time consuming) such as for the following path and predicate expression:

```
//department[@street="main mall"]/student[@nationality="Chinese"]
```

It is very likely that “department” is not indexed on “street” and that “student” is not indexed on “nationality”. Therefore, resolving this path expression will cause table scans. Moreover, storing XML data in an RDBMS often results in schema decomposition and produces many small tables. Hence, evaluating a query often needs many expensive join operations.

For unstructured or semi-structured data, an RDBMS has greater difficulty, and query performance is usually unacceptable for relatively large amount of data. For these reasons, a native database management system is expected in the XML world. Like a traditional RDBMS, native XML databases would provide a comprehensive and generic data management interface, and therefore isolate lower level details from the database applications. Unlike an RDBMS, an ideal native XML

database would make no distinction between unstructured data and strictly structured data. It treats all valid XML data in the same way and manages them equally efficiently. Its performance is only affected by the type of data manipulation. In other words, an ideal XML native database is not only access transparent but also performance transparent upon the structural difference of data.

Compared to an RDBMS, XML database techniques are far from mature. Motivated by the desire of developing such a native XML database, this thesis, at a low level, implemented a native XML data storage and index structure that is particularly tailored for XML data and completely hidden from end-users. At a high level, this work implemented the XQuery 1.0 language drafted by W3C [26]. The process of building the storage and index structure is automatic and generic because it relies neither on XML schema existence, nor on the participation of database administrators or database server-side programmers. It presents a uniform data manipulation interface, regardless of the structure of data, and therefore is access transparent. The entire system was measured in three aspects: query evaluation performance, storage size, and time to build the indices. Our performance evaluation shows that it is also performance transparent.

The rest of this thesis is organized as follows. Because it is very easy to represent objects in XML, and the nested structure of XML data can be naturally viewed as object compositions, object-oriented techniques are discussed in Chapter 2. In fact, this work also adopts the object view of XML data and uses the term “object” to represent the information carried by an XML data chunk. In Chapter 3, we choose the Oracle XML Database as an example for a thorough case study: we introduce the new technologies widely-adopted in the relational database industry to manipulate data as objects; we also compare the main features of this product

with the desired features of an ideal native XML database. In Chapter 4, we briefly introduce some native XML database products. These products are called “native” because they are not built on top of any existing system. After that, we introduce the XQuery and XPath languages and their underlying data model in Chapter 5. Finally, in Chapter 6, we briefly describe the system architecture, core functionalities, and performance tests of this native XML database implementation.

Chapter 2

XML Data As Objects

2.1 XML Data as Objects

While it is very easy to represent objects in XML, it is also natural to view XML data as objects. The nested structure in XML data is a natural mapping of object composition in the object-oriented programming world. Since people resort to object-oriented databases for an object repository directly connected to programming applications, object-oriented databases are intuitive solutions for persistent XML objects storage.

On the other hand, there are also good reasons that prevent object-oriented databases from being an ideal database solution for XML data. First, in object-oriented programming, the application context is usually fixed or very stable. Class prototypes and definitions precisely capture all characteristics of objects in that context. Object-oriented databases can then take advantage of the class information and allocate just enough resources to store and manage objects on disk. However, for XML data, this implies stable schema existence and less flexibility of adapting structural difference.

Second, in object-oriented programming, we don't access objects arbitrarily. Instead, we typically access them through the open interface designed by application programmers. The application programming context often makes it obvious to choose indices to facilitate efficient operation through the open interface. For example, a financial application may allow its customers to obtain a reference to their account objects only by the following public method:

```
public Account& getAccount (const string& name,  
                             const string& password);
```

Then it may be a good idea to create an index on the concatenation of *name*, “;” and *password* for all customer account objects on the disk. However, in XML, we typically don't have a fixed functional interface to access the data. On the contrary, we would like to give database end users more control to ask various queries, as long as they are valid in terms of the syntax and semantics of the query language. The absence of knowing how data will be queried often leads to unacceptable query performance if XML data is stored in object-oriented databases.

2.2 Document Object Model

In the Document Object Model [20] [21] [22], the term “document” is not bound to conventional text documents, but rather it refers to any kind of information that is represented in XML format. “Increasingly, XML is being used as a way of representing many different kinds of information that may be stored in diverse systems, and much of this would traditionally be seen as data rather than as documents” [20].

DOM is an object-oriented abstraction of XML documents. First, it repre-

sents an XML document as a tree object that contains exactly one document root node, which in turn is composed of node objects of other types. The hierarchical node composition relationship defined in DOM is shown in Table 2.1.

Node Type	Composite Member Objects
Document	Element (maximum of one), ProcessingInstruction, Comment, DocumentType (maximum of one)
DocumentFragment	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
DocumentType	no children
EntityReference	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
Element	Element, Text, Comment, ProcessingInstruction, CDATASection, EntityReference
Attribute	Text, EntityReference
ProcessingInstruction	no children
Comment	no children
Text	no children
CDATASection	no children
Entity	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
Notation	no children

Table 2.1: Composition Relationship of DOM Node Objects

Second, DOM defines a standard mechanism to access and manipulate the content of document objects and their component objects. In other words, DOM defines a set of uniform functional interfaces to read or modify the content of XML documents. Therefore, DOM is essentially a language neutral programming interface. Following the class hierarchy idea used in object-oriented programming, DOM organizes the functional interface by different levels of abstraction. For example, the DOM “Node” interface is an abstraction of all node object types in Table 2.1, and declares the common methods to manipulate these objects. As a subtype of “Node”, the DOM “Element” interface inherits all methods of the “Node” interface, and has

its own additional ones. Besides the individual object-based functional interface, DOM also provides several container type abstractions to represent collections of “Node” objects in documents. The “NodeList” interface represents a collection of ordered “Node” objects. The “NamedNodeMap” interface represents the associative relationship between names and collections of “Node” objects.

2.3 DOM Persistence on Disk

There have been a number of DOM implementations in different programming languages for different purposes. Some implementations are embedded in application software. Microsoft Internet Explorer 5, for example, deals with HTML documents or XML documents. Some implementations provide programming language library packages to access and update the content of HTML or XML documents (e.g., the Java extension DOM API). Others serve as the middleware interface between applications and XML database systems, or between different software layers of the same XML database system. An example in this category is Oracle XML DB, which constructs DOM tree objects to support query evaluation for those XML documents stored in its unstructured storage.

The first two types of DOM engines are dedicated to handling relatively small XML documents in memory. The third type of DOM implementations are incorporated in some XML database systems that have persistent storage and are capable of handling large XML documents. Nevertheless, the DOM engine itself does not provide persistent storage, nor does it directly manipulate data on disk. In contrast to the first two types of DOM implementations, DOM Persistence on Disk directly maps DOM objects to data blocks on disk and manipulates them through the DOM interface. The major advantage of this approach is that XML data can be

stored into an existing object-oriented database. And the fixed functional interface of DOM presents enough information to create indices, as we have described in Section 2.1.

eXtc is an XML database released by M/Gateway Developments Ltd., following the DOM Persistence on Disk idea. It is built on top of the Caché database system and is implemented in Caché's ObjectScript programming language. Caché is a database product claiming to be a post-relational database with both object-oriented and relational features, by InterSystems Corporation. Like traditional object-oriented databases, Caché is tightly coupled with its target object-oriented programming language, namely, Caché's ObjectScript language. For objects that need persistent storage, Caché provides its own language syntax to define the class prototypes. On the basis of the class prototype, Caché's class compiler automatically organizes the physical structure to store these objects, which are called "global" (named multi-dimensional arrays) to store persistent data. eXtc chose Caché as the underlying database and implemented the DOM interface in Caché's ObjectScript language. The implementation work includes class prototype declaration of DOM objects and DOM method definition in ObjectScript language. Each DOM interface method directly manipulates data on disk through Caché.

In addition to the standard DOM interface, eXtc also supports a portion of XPath semantics, more specifically the path expression. While eXtc achieves the basic functionalities of an XML database such as persistent storage and a data manipulation interface, there are some major limitations. First, eXtc suffers the query expressiveness limitation of the DOM interface. Second, although it is possible to execute every method in the DOM interface very fast, more complicated methods implemented on top of the DOM interface could be very inefficient. As an example,

a path expression containing a descendant axis will result in an exhaustive search on disk.

2.4 SQL:1999, Object-Relational Model

SQL:1999 [16], also known as “object-oriented SQL”, is intended to become the new SQL standard with both relational and object-oriented features. It introduces the object concept and additional language syntax to manipulate complex objects in the same way as other built-in data type instances. On the basis of SQL:1999, many RDBMS vendors (e.g., Oracle, IBM) have enhanced their products with object-oriented support in recent years. These products are sometimes called object-relational databases. Unlike object-oriented databases, object-relational databases are not bound to programming languages. By implementing the language enhancement introduced in SQL:1999, these products inherit the SQL interface and therefore achieve access transparency. In other words, the functional enrichment of handling objects in SQL does not require additional skills on DBAs and database application programmers.

Since it is natural to view XML data as objects, as we have described in Section 2.1, object-relational databases are also good for managing XML data. We postpone our analysis of this approach to next section, where we do a thorough case study on Oracle XML DB, introduce its working mechanism, and compare its features to the desired features of an ideal XML database solution. Here, we only introduce the major object-oriented enhancements of SQL:1999.

2.4.1 User-defined Data Types

To capture the complex structural information encapsulated in objects, SQL:1999 introduced a new language syntax to abstract the prototypes of objects. Similar to class declarations in object-oriented programming languages, SQL:1999 allows users to specify the composite data members of a new data type, the type inheritance relationship to an existing data type, and the common behavior of all objects of this new data type.

This SQL:1999 syntax example (taken from [16]) illustrates SQL:1999 new syntax to define a new data type *emptytype* under inheritance:

```
CREATE TYPE emptytype UNDER person_type
AS ( EMPID      INTEGER,
      SALARY    REAL )
INSTANTIABLE
NOT FINAL
REF ( EMPID )
INSTANCE METHOD
      GIVE_RAISE ( ABS_OR_PCT  BOOLEAN,
                  AMOUNT      REAL )
                  RETURNS REAL
```

It corresponds to the following Java class prototype.

```
public class emptytype extends person_type
{
    public int    EMPID;
    public double SALARY;
```

```

    public double GIVE_RAISE (boolean ABS_OR_PCT, double AMOUNT)
}

```

In addition, *INSTANTIABLE* corresponds to the absence of the Java *abstract* modifier in the class header, and *NOT FINAL* corresponds to the absence of the Java *final* modifier in the class header. *REF (EMPID)* doesn't have a syntactic counterpart in Java. Here, it means the *EMPID* field has a unique value for each *emptytype* object.

The new user-defined data type can be used in the same way as a common SQL built-in data type (e.g., the data type of a column in a table, the data type of host variables, or the data type of a field in some other user-defined data types).

2.4.2 Object References

The *REF* keyword can be also used as references to objects of a pre-defined data type. For example (taken from [16]), if *emptytype* has been defined as a new data type, we can create a table that contains a *manager* column. The data type of the *manager* column could have the notation *REF(emptytype)*, which implies references to objects of *emptytype*.

As in object-oriented programming languages, references in SQL:1999 can either point to real objects or hold null references. Unlike references in object-oriented programming languages, *REF* doesn't have *lvalue* semantics. The only way to modify the attribute values of an object is through the system generated "setX" functions. In addition, there are no different levels of access control. All attributes of objects are "public" accessible.

Chapter 3

A Case Study: Oracle XML DB

We chose Oracle XML DB as an example of storing XML data in an RDBMS because we found it has more comprehensive and advanced XML features than those from other RDBMS products. For convenience, Oracle XML DB is abbreviated as OXDB in the remainder of this thesis.

From a logical point of view, OXDB “fully absorbs the W3C XML data model into the Oracle Server” [27] because XML data can be manipulated through a relatively native interface rather than a relational interface (e.g., SQL statements integrated with XPath expressions instead of purely set operation based SQL statements). The design goal of OXDB can be summarized as achieving access transparency by the provision of a native XML data manipulation interface and the isolation of mapping XML documents to relational tables from database client users. This summarization is further explained in the rest of this chapter. The functionalities OXDB actually achieves are compared with the corresponding features desired in an ideal native XML database.

3.1 Storage

OXDB introduces a new “XMLType” data type to represent arbitrary XML content from an open tag to its balanced close tag. Thus, an identifier (variable) of type XMLType can contain one or more trees of well-formed XML data. Like other built-in data types, this XMLType can be used to specify the data type of a column in a table. It is also possible to create a table solely containing XMLType instances. On absence of XML schemas, XMLType objects can only be stored as Character Large Objects (CLOB), which are basically text strings. When schemas are available, an XMLType object can be decomposed into a set of SQL data type objects. In this case, an XMLType object can be stored as either a CLOB object or a set of smaller SQL objects.

The new XMLType data type successfully achieves access transparency because both strictly-structured and unstructured XML data can be stored in the database using the same SQL command. Upon completion of schema registrations, client side users are completely isolated from the actual details of how their XML documents are physically stored. The client side application code works for both the unstructured XML data and the structured XML data without the need of modification.

However, this approach is not *performance* transparent. Client side users can easily observe the throughput difference for document insertion and retrieval operations. The cost of processing large object decomposition and integration is very expensive especially when the object components scatter on disk, such as when they are indexed and clustered by some attribute value. For many applications (e.g., on-line transaction applications), performance fluctuation is certainly undesired. An ideal native XML storage, on the other hand, is expected to be both access

transparent and performance transparent.

3.2 Schema Mapping and Table Creation

OXDB automates the schema mapping process by providing the built-in PL/SQL procedure “`dbms-xmlschema.registerschema()`”. Upon successfully registering an XML schema, OXDB automatically creates a default table of XMLType objects, in which each row is used to store a valid XML document. The schema mapping problem is solved by deriving new complex data types to represent the document, elements and attributes declared in the schema. A nested table may be created for a collection of objects appearing in an XML document. A valid XML document can be loaded into the repository using an SQL or PL/SQL insertion statement, or by using FTP, HTTP or WebDAV applications. When the schema is not available, OXDB makes no attempt to create a relational table. XML documents are simply stored as text strings. OXDB also supports schema evolution. But updating a schema requires all existing XML documents to be compliant with the new schema, and a complete copy of all documents is made in the schema updating process.

However, this schema-mapping process is less flexible and adaptive than the desired features of an ideal native XML database. It is usually the case that people want to put variety of objects that logically belong to the same class into the same storage location. For example, suppose a printer dealer started his business by selling black ink printers. Some time later, laser printers and color printers became available, but this printer dealer would have found no way to fit these new types of printers into his black-ink printer table, assuming OXDB techniques were used. Putting black-ink printers, laser printers and color printers into the same storage location simplifies the database management task. More importantly, queries

regarding the printer concept hierarchy can be evaluated by looking at a single table.

OXDB has some difficulties in supporting this kind of concept hierarchy, partly because a single table cannot be compatible with all subtypes. More importantly, relational databases suffer the problem of viewing data at a static point. Ideal XML native databases would take care of the concept hierarchy problem gracefully because concept hierarchies can be modeled as DOM trees. Introducing a new subtype is simply done by adding a tree node. Storing objects of different subtypes in a database amounts to adding sub-forests.

Next, it is not genuinely automatic and generic to perform the schema-mapping and table creation task in OXDB. We require non-trivial efforts at the database server side to alter the default table that is automatically created by schema registration. Such effort includes creating/dropping indices, determining whether to store a collection of elements in a nested table, re-naming new derived data types, etc. All of these require the participation of server side programmers or database administrators. Furthermore, the application context must be considered when making a decision on these issues. Moreover, it is hard to predict the amount of server-side effort when the application workload changes and scalability is required.

3.3 Index and Query Evaluation

OXDB brings a hierarchical index into the XML data repository. This hierarchical index is transparent to end users, including server-side programmers and DBAs. It allows XML content to be represented “as documents in a folder hierarchy, and hierarchical metaphors, such as paths and URLs, to access documents and represent the relationship between documents” [28]. Some previous studies on hierarchical

indices include [1] and [8].

After schema registration, a DBA may add a unique constraint to an element, attribute, or combination these of using an SQL statement. The uniqueness applies to all documents in the default table that is automatically created by schema registration. This constraint declaration implicitly builds a B-Tree index on the default document table.

In addition, a DBA can explicitly create two other types of indices on XML content using SQL statements. First, XPath expressions can be embedded in index creation statements. For XML documents stored in structured storage, OXDB will first use a query-rewrite technique to determine if the XPath expression can be restated using Object Relational SQL [28]. If this is the case, the index is created as a conventional B-Tree index on the underlying SQL objects [28]. Otherwise, a functional index is created for this XPath expression. In other words, OXDB assumes the XPath expression that is used to create an index will be frequently asked at query time. Therefore, it caches the question and its pre-computed answer. Second, two text indices can be created for XML documents: (1) OXDB's "ctxsys.ctxxpath" text index speeds up answering a question like "Does this string appear in this document?", and (2) OXDB's "ctxsys.context" text index is used for full text search. XML documents stored as CLOBs can only have functional indices and text indices.

For queries containing XPath expressions, OXDB can optimize the query execution plan when B-Tree indices or functional indices are available on some query fields. If none of them are created but a text index exists, OXDB uses the text index as a filter condition so as to avoid evaluating the query for each document in the table. The implementation details of the text index are not given in Oracle's documentation, to the best of our knowledge. Therefore, it is not clear whether the

text index can eliminate full table scans. Assuming it does, a significant number of disk seeks is still foreseeable. If there is no index to help answer the query, then a full table scan is unavoidable. For XML documents stored as CLOB objects, query evaluation will cause the construction of a DOM tree in memory for each XML document.

A B-Tree index, functional index and text index provide relatively flexible control on query performance tuning. It is access transparent for the client side because a valid XPath expression will eventually produce the same result, regardless of what indices are built at the database server side. However, a query execution plan requiring a full table scan may be acceptable when there are only a few hundred documents in the table, but would be unacceptable if there are thousands or millions of documents in the table [28].

3.4 Conclusion

OXDB brings comprehensive native support to XML data for relational databases. Such support is access transparent but not performance transparent. Like a conventional RDBMS, there is non-trivial work at the database server side to address the database design and performance tuning issues. Making a decision on these issues requires not only the participation of server-side programmers and DBAs, but also enough knowledge about the application domain. Therefore, OXDB does not genuinely present an automatic and generic process of moving XML data to database management systems.

For relational data represented in XML format (e.g., data transformed from an existing RDBMS), OXDB provides high performance and some degree of automation if a schema exists. It is very expensive to update a schema and very

difficult to ask queries in terms of a concept hierarchy. For non-relational data, query performance is unpredictable at the database client side.

This case study of choosing Oracle XML DB as an example also makes it clear that the current RDBMS techniques are capable of providing acceptable solutions in many business domains, especially when XML data is highly structured or data size is relatively small.

Chapter 4

Native Works

We have discussed storing XML in object-oriented databases and relational databases. In this chapter, we introduce some existing native XML database products. Many other products also claim that they are native because they support XML data manipulation in a native way such as via the XPath language or the DOM interface. However, these products are directly built on top of an existing database system (RDBMS or object-oriented DBMS) and are essentially database middleware products with functionalities for manipulating XML data. Therefore, these products inherit flexibility and efficiency limitations from their underlying systems as we have discussed in previous chapters. Here, we use the term “native” referring to those systems that not only provide a native functional interface, but also physically manipulate XML in their own way. In other words, those systems do not map XML data to objects nor to relational tuples. Moreover, the term *native* does not mean that existing techniques cannot be used. Instead, a native product is free to borrow any techniques from an existing system as long as it does not take an existing system as the whole back-end.

4.0.1 Berkeley DB XML

Berkeley DB XML [31] is an open source database library supporting the XPath 1.0 language. It relies on the Berkeley DB database library for basic database functionalities (e.g., storage, update, concurrency control). XML documents are directly stored in the database without any conversion. The logical unit of XML document storage is a *Container*, in which one or more XML documents can be stored. The *Container* class has member functions as the programming interface to add, delete and replace documents.

Programmers can create indices for all documents in a *Container*. An index can be built on XML nodes, edges, node types and key values. While defining these indices, users also need to specify the semantics of these indices (e.g., comparing them as string, just checking the node existence, etc.). An index can be added, deleted or replaced. Programmers can embed XPath query into their code and run the query through Berkeley DB XML's API.

4.0.2 eXist

eXist [34] is a Java open source XML database with XQuery support. The most interesting design idea of this product that we believe is essential to ideal native XML databases is automatic indexing. After users have set up the database server side configuration, eXist will automatically build indices on the incoming XML documents to be stored. eXist by default builds two types of indices. Like a traditional search engine, eXist builds a full-text index on XML document content, with pruning of stop words. In addition, eXist treats an XML document as a DOM tree and builds a node index for top-N level element nodes in the DOM tree, where N is set in the database configuration. Axis relationships such as parent-child, ancestor-

descendant, are resolved by node indices.

Unfortunately, eXist also has a number of limitations. For example, one important distinction between a database and a search engine is the precision of results. Consider the simple query `//article[author='Marx, Karl']` taken from eXist on-line documentation [35]. This query evaluation will be very slow unless it is reformulated to `//article[near(author, 'Marx, Karl')]`. However, the latter is unacceptable in the database world because it may return the author being 'Karl, Marx', 'Mar Kar', ..., or even no result at all if `Marx` and `Karl` are stop words. In addition, the way eXist builds element node indices makes it neither flexible, because it is not easy to modify the structure of a document that has been indexed, nor performance transparent, because query evaluation highly depends on where an element node involved in the query is actually located in the DOM tree.

4.0.3 Apache Xindice

Xindice [36] is a Java open source database developed by the Apache XML project group. It implemented XPath and XML:DB XUpdate to query and update XML documents. Different indices can be built or deleted manually through the command line interface. An index can be on the content of some leaf elements or all leaf elements, and on the attribute values of some elements or all elements.

Xindice is designed to handle small or medium sized documents. XML documents are stored in collections in hierarchical structures like a traditional file system. For example, one collection can contain some sub-collections. Each document is associated with a unique id when it is added into the database. Data manipulation is done at the unit of individual documents. In other words, a query or update on an arbitrary document fragment is not supported.

Chapter 5

XQuery and XPath Language

5.1 XQuery and XPath Data Model

The XQuery and XPath data model [23] is a tree-based model for XML documents, in which there are seven different kinds of nodes. Each node instance is distinct from any other node instance. If the root node of the tree is of document-node type, then this tree represents a complete XML document, otherwise this tree represents a fragment of the document.

All information accessible in this data model is in the form of sequences. In other words, the input and output of XQuery and XPath programs are sequences. And all intermediate results in query processing are also sequences. Each sequence contains zero or more items. Duplicate items may occur in a sequence. Each item is either an atomic item or a node item. A single item is equivalent to a singleton sequence. Because one sequence cannot be nested inside another sequence as its member item, the structure of sequences is flat.

5.1.1 The Atomic Items

An atomic item is a value instance of an atomic type. There are 21 built-in primitive atomic data types as listed in Table 5.1. *untypedAtomic* and *anyAtomicType* (No. 20 and No. 21 in Table 5.1) are defined in this data model, and all others (No. 1 to No. 19 in Table 5.1) are defined in the XML Schema [24].

In addition, a subtype of an atomic data type is also an atomic type. The built-in subtypes derived from these primitive atomic types are also listed in Table 5.1. However, a new data type by composition of existing atomic types is not an atomic type.

5.1.2 Data Type and Naming

On the basis of the built-in atomic data types described in Section 5.1.1, we can define a new atomic data type by sub-typing an existing atomic type, and we can define a new complex data type either by listing finite-length values of an existing atomic type, or by the union of one or more existing data types (which may or may not be atomic).

Each data type must have a unique type name, which includes a namespace prefix and a unique type name in that namespace. All built-in data types defined in XML Schema, including atomic types (No. 1 to No. 19 in Table 5.1) and non-atomic types, have global scope names with namespace prefix *xs*. All built-in types defined in this data model have namespace prefix *xdt*. User-defined data types in their schema definition may be explicitly named under the user's namespace prefix or implicitly be given a unique anonymous type name by the actual implementation of the data model.

No.	Primitive Type	Immediate or Non-immediate Built-in Subtypes
1	string	normalizedString, token, language, name, NMTOKEN, NCName, ID, IDREF, ENTITY
2	boolean	none
3	decimal	integer, nonPositiveInteger, long, nonNegativeInteger, negativeInteger, int, unsignedLong, positiveInteger, short, unsignedInt, byte, unsignedShort, unsignedByte
4	float	none
5	double	none
6	duration	dayTimeDuration, yearMonthDuration
7	dateTime	none
8	time	none
9	date	none
10	gYearMonth	none
11	gYear	none
12	gMonthDay	none
13	gDay	none
14	gMonth	none
15	hexBinary	none
16	base64Binary	none
17	anyURI	none
18	QName	none
19	NOTATION	none
20	untypedAtomic	none
21	anyAtomicType	none

Table 5.1: Built-in Atomic Data Types in XQuery and XPath Data Model

5.1.3 The Node Items

In this data model, a “node” serves two purposes. First, a node abstracts a fragment of an XML document in the tree representation. Second, a node has its associated semantic values and possibly some data type bound to it. When nodes appear as items in sequences, their semantic values and data type information form the input, output and intermediate results of query processing.

There are seven different kinds of nodes as listed in Table 5.2. All of them can be leaf nodes in the tree representation. All of them have their associated string-

values. For example, the string-value of a document node is the content of the entire document. In addition, the element nodes and attribute nodes are also bound to particular data types and therefore carry typed-values.

Node Type	Child Node Types
Document	Element, Processing Instruction, Comment and Text (non-empty)
Element	Element, Processing Instruction, Comment and Text (non-empty)
Attribute	Empty
Namespace	Empty
Processing Instruction	Empty
Comment	Empty
Text	Empty

Table 5.2: Seven XML Node Types in XQuery and XPath Data Model

5.2 The XPath Language

XPath [25] is a strongly-typed query language on the basis of the XQuery and XPath data model. The primary non-terminals of this language are expressions. An entire XPath query program can be fully reduced to a single expression according to its grammar production rules. Expressions are defined by terminal tokens and operand expressions, recursively.

All expressions have sequences as their return values. The data type of sequences has the form: *ItemType OccurrenceIndicator*, where *ItemType* is the type name of items in this sequence, and *OccurrenceIndicator* could be “?” representing optional single item, or “*” representing zero or more items, or “+” representing one or more items. The declared data type of an expression must match the expected return type in accordance with the corresponding semantic rules. The declared type of an expression must also match the type of its actual return value.

Expression evaluation depends on two types of context information: static and dynamic. Static context contains all information available before the evaluation of the entire query program, such as the built-in data types. Dynamic context covers all information accumulated up to the point where an expression is evaluated such as the run-time value of a variable.

5.3 The XQuery Language

XQuery [26] is an extension of the XPath language, sometimes called a superset of XPath. The most important extensions are the following:

1. XQuery introduces module definitions. A module can be a main module, which is a complete query program, or a library module that exports library functions and variables.
2. To facilitate more flexible control on query evaluation context, an XQuery module may contain a Prolog definition. Programmers can choose default namespaces at query time, import pre-defined schemas and library modules, bind global variables to some values, and define global functions.
3. A type-switch expression is a run-time dynamic type checking mechanism. Depending on the dynamic data type of an operand expression, a type-switch expression evaluates one of its case expressions and returns the expression as its own return value.
4. A constructor expression constructs an XML document fragment inside the query body. Query programmers can restructure XML data and produce different views.

Chapter 6

This Native XML Database

6.1 System Architecture

6.1.1 Overview

Motivated by the “generic” and “automatic” design goals of an ideal XML database, this work consists of two major parts: XML data storage and XQuery evaluation, shown in Figure 6.1 at the end of this chapter. Each part is a separate library to be included in application-specific processes or a general purpose database management system. These two libraries can also be viewed as the functional interface open to database programmers.

6.1.2 The Data Storage Part

In this system, XML data objects are physically stored in collections. A collection is a schema-insensitive storage place and therefore accepts any well-formed XML data. The API of this system provides four functions allowing users to create collections and adding objects into collections, as listed in Table 6.1. It is completely up to the database programmer to decide into which collection an object should be added.

Function Prototype	Usage
<code>bool existCollection (const string& c_nm)</code>	return true if the database contains a collection with the given name
<code>void createCollection (const string& c_nm)</code>	create a new collection
<code>void swtichCollection (const string& c_nm)</code>	switch to a particular collection as the default working collection
<code>void addObjects (istream& obj_in, const string& c_nm = "")</code>	Add objects to a collection. On absence of the second parameter, add them to the default working collection.

Table 6.1: Server-side API of This Native Database

The *DatabaseManager* class has a fixed number of active collections loaded in memory, depending on the actual system configuration. When the *DatabaseManager* is requested to add some objects into a particular collection, it checks if the target collection is active. If not, it prepares an active collection by opening necessary index files and log files and also loading some meta data for that collection. Then the *DatabaseManager* delivers the active collection and the XML input stream passed as a parameter of the `addObjects` function to the *IndexManager*. The *IndexManager* processes the meta data of this collection and creates an *Indexer* agent to build indices. Then it passes the XML data input and the *Indexer* agent to the *XML parser*. The *XML parser* parses all information from the first open tag to its matching close tag as the data belonging to one object. If the input is not exhausted, it treats the subsequent data as additional objects. During the parsing process, the *XML parser* invokes the *Indexer* agent to insert different indices into appropriate index files. In any erroneous circumstances, an exception is thrown with an appropriate error code.

6.1.3 The Query Evaluation Part

This XQuery implementation is independent of the actual details of how XML data objects are physically stored on disk. Like an implementation of some general purpose programming language that needs system call services for a particular machine platform, this XQuery implementation relies on a set of engine calls as the uniform functional interface that the underlying data storage will support. These uniform engine calls are declared as pure virtual functions encapsulated in the *DataEngine* super class. A storage-specific subtype of the *DataEngine* class will decide how to actually implement these engine calls most efficiently.

The *DatabaseManager* class serves as the storage-specific data engine generator in the query evaluation package. It creates a *NativeDataEngine* object, a subtype of *DataEngine*, through its `getDataEngine(const string& base_uri)` member function. This native data engine object will take the `base_uri` parameter in the `getDataEngine` function as the base URI to resolve all relative locations of resources.

The interface open to users in the query evaluation package includes the following functions, where XQ is the short alias of this XQuery implementation namespace.

```
XQ::Program* getXqueryProgram (XQ::DataEngine* engine, const char* src);  
void XQ::Program::decorate ();  
XQ::Sequence* XQ::Program::eval ();
```

The global function `getXqueryProgram` takes a storage-specific data engine object and a string buffer of the actual XQuery code as parameters and returns an XQuery program object. This XQuery program is represented as an abstract syntax

tree and will be evaluated on top of the given data engine. The `decorate` member function of the `XQ::Program` class performs the task of static type checking and symbol resolution in three steps:

1. It will create a static XQuery evaluation context by importing the XQuery built-in data types and operators.
2. It will import all uniform engine calls from the given data engine object.
3. It will decorate the user-defined data types, variables and functions declared in the actual XQuery code.

Upon completion of these three steps, a symbol table is created with all names properly resolved, and the XQuery program has successfully passed static type checking. Finally, the `eval` member function of the `XQ::Program` class will evaluate the query program and return the result sequence. During the evaluation process, dynamic type checking is performed to make sure the return data types of XQuery expressions match their expected or declared data types. In any erroneous circumstances, a static or dynamic type-checking exception is raised with an error code.

6.2 Database Design

A variety of objects may be stored in the same storage place. The database design task in this system is simplified as defining logical concepts and classifying objects into collections. As an example, a printer dealer may design his database containing a collection of printers. Under the general concept “printer”, there may be “color printer” and “laser printer” subtypes. Though printer objects of different types have

different schemas, they can be added into the same collection. While it is completely up to the database designer to decide the association relationship between objects and collections, there are two general design principles:

1. Objects in the same collection need not have identical structure, but a collection should contain objects that logically belong to the same class or are closely related to each other.
2. The design choices of associating objects to collections should not require object ordering because objects in a collection are unordered.

The first principle is not obligatory and is solely for the purpose of query efficiency and ease of database management. Although the system does support queries across different collections, it can only maintain a fixed number of active collections in memory. If it just happens that the query is on an inactive collection, the database manager needs to evict one currently active collection and bring the collection requested into memory. This involves opening all index files and log files and loading meta data from disk. Therefore putting closely related objects in the same collection will make query evaluation faster, and will make the database management task easier.

On the other hand, the second principle must be satisfied for correctness purposes. One important distinction between XML data and relational data tuples is the data order. In XML documents, the document order is one part of the information carried by the data. While in relational tables, the order of different tuples is typically neither of interest, nor essential to application correctness. Superficially, the second guideline causes the loss of document order information. However, one important observation is that document order is just a partial order restricted in the

local context. The order of objects in a different context is usually not important. As a concrete example, all information contained in a book needs to maintain its original order, but the actual order of books on a shelf or the order of two chapters in two different books might be not important. In that case, the database designer should create a collection of books and associate complete book objects, rather than chapters, to this collection. And this does not cause any inconvenience of retrieving a fragment of a book.

6.3 Genericness and Automation

This system is application-independent in the sense that it does not rely on the structure of the target XML data, nor on XML schema existence. Any well-formed XML data will be accepted by the system and yet lead to transparent performance. It is also highly automatic because the system does not request the participation of DBAs or database server side programmers to make many database physical design decisions that are inevitable in traditional RDBMS, such as creating/modifying tables, building/dropping indices, etc. The database design task in this system is more focused on logical design.

The underlying index structure is completely hidden from users and is automatically built via internal method invocations in the data storage package. The end users need not be aware that their XML data has been indexed. Instead, the system presents a native XML tree view to users, where the collection is the root node of the tree. Whenever a user passes a well-formed XML data chunk to the *DatabaseManager*, the system extracts all outermost elements from the input stream and stores them as child nodes of the collection root. Document order is preserved in the local context of each outermost element. But the relative order of these out-

ermost elements is not guaranteed. In other words, users can put multiple XML documents into one input stream and the system will treat each complete XML document as an outermost element, which should contain all information describing a complete object. Queries are asked solely on the basis of this native tree view of XML data without necessity of knowing the physical structure of the storage.

The current version of this system does not support schema importation. Therefore, it does not perform validation checking on the input objects. However, the system design of the current implementation leaves a door open to schema importation in future versions. Namely, the system is open to schema definitions as new data types on the basis of existing data types. Upon completion of supporting schema importation, the system will not only be able to validate XML data objects, but also be able to associate objects with a more precise concept that best describes them. Again, taking the printer dealer example, the system will import a “color printer” schema and construct a concept hierarchy with “color printer” as a subtype of the general “printer” type. Then all color printer objects are instances of both these printer types. Users can ask queries in terms of the logical concept hierarchy.

6.4 Performance Measurement

We populated 5 XML documents containing 500, 1000, 2000, 4000, and 8000 vehicle objects as the test data. The attribute values to describe each vehicle object are randomly drawn from their domain values. The document size starts from 64 KB (500 objects) up to roughly 1 MB (8000 objects). The experiment was carried out on a Toshiba laptop with a Celeron 600MHz CPU, 64 MB RAM and Windows 98 operating system. We performed experiments on the test data to measure the total physical size of data storage including indices, the time to build indices, and the

time to evaluate path expressions with different path lengths.

Neither the data storage nor the query evaluation part is yet optimized in the current implementation. The total physical storage size grows linearly with document size. Figure 6.2 compares total storage size to document size. Total storage size is about 8-9 times the size of the original document. Although the graph trend suggests that the ratio decreases with document size, it is only a small decrease in real terms. The time to build indices has better scalability than linear growth, and is shown in Figure 6.3. We ran query evaluation tests of path expressions with path lengths increasing from 1 to 5 on different numbers of objects. Figure 6.4 shows that query evaluation time grows linearly with the path length and the total number of objects in storage.

6.5 Summary

We believe an ideal native XML database needs to be automatic and generic. Our implementation treats any well-formed XML data objects in the same way regardless of their structures. It automatically puts objects into object collections and builds indices just on the basis of data itself. It neither relies on schema existence nor on application domain knowledge. Our query performance evaluation shows linear scalability when the number of objects in a collection increases and when the length of a path expression increases. Even though the schema information is absent in both the object storing process and the query evaluation process, the overall query running time is acceptable for a relatively large number of objects.

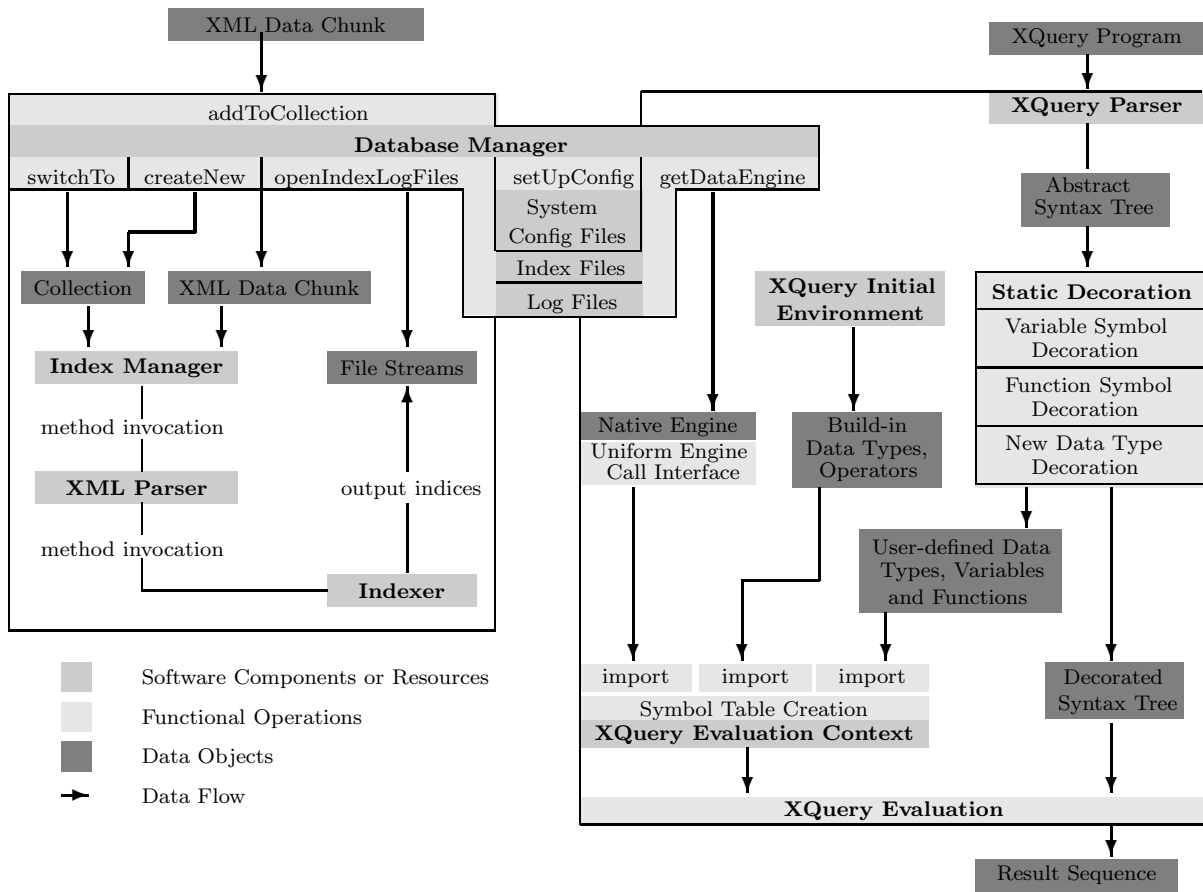


Figure 6.1: System Architecture of This Native Database

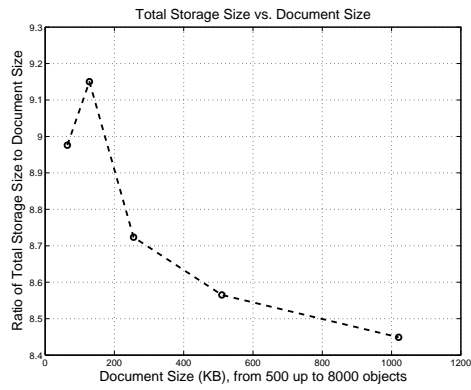


Figure 6.2: Storage Size vs. Document Size

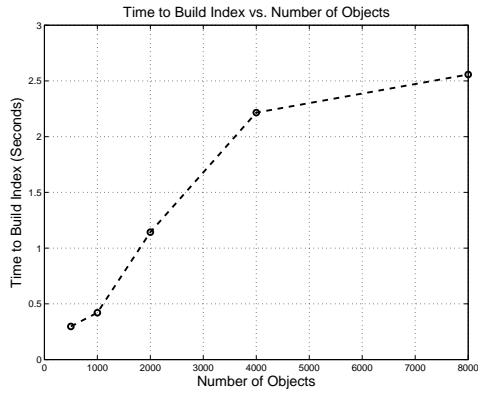


Figure 6.3: Time to Build Indices vs. Number of Objects

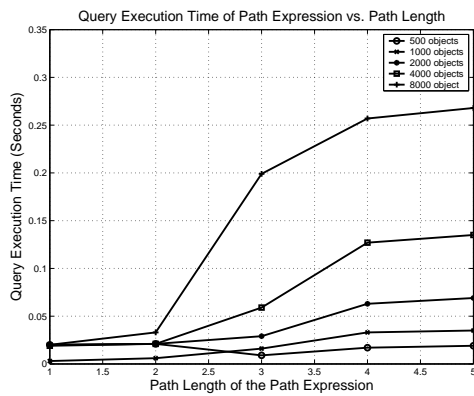


Figure 6.4: Query Time vs. Path Length and Number of Objects

Bibliography

- [1] B.F. Cooper, et al., “A Fast Index for Semistructured Data”, *Proc. VLDB*, 2001, pp. 341-350.
- [2] C.Y. Chan, M. Garofalakis and R. Rastogi, “RE-Tree: An Efficient Index Structure for Regular Expressions”, *Proc. VLDB*, 2002, pp. 251-262.
- [3] C.C. Kanne and G. Moerkotte, “Efficient Storage of XML Data”, *Proc. ICDE*, 2000, pp. 198.
- [4] Y. Yamane, N. Igata and I. Namba, “High-performance XML Storage/Retrieval System”, *FUJITSU Sci. Tech. J.* 2000, pp. 185-192.
- [5] R. Baeza-Yates and G. Navarro, “Integrating Contents and Structure in Text Retrieval”, *ACM SIGMOD Record* 25, 1(Mar.), pp. 67-79.
- [6] G. Navarro and R. Baeza-Yates, “Proximal Nodes: A Model to Query Document Databases by Content and Structure”, *ACM Transactions on Information Systems*, Vol. 15, No. 4, October 1997, pp. 400-435.
- [7] S. Al-Khalifa, et al., “Structural Joins: A Primitive for Efficient XML Query Pattern Matching”, *Proc. ICDE*, 2002, pp. 141-152.

- [8] T. Milo and D. Suciu, "Index Structure for Path Expressions" *Proc. ICDT*, 1999, pp. 277-295.
- [9] C.L.A. Clarke, et al. "An Algebra For Structured Text Search and A Framework for its Implementation", *The Computer Journal*, 1995, pp. 43-56.
- [10] M. Consens and T. Milo, "Algebras for Querying Text Regions" *Journal of Computer and System Sciences*, 1998, pp. 272-288.
- [11] M. Consens and T. Milo, "Optimizing Queries on Files", *Proc. ACM SIGMOD*, 1994, pp. 301-312.
- [12] S. Abiteboul, S. Cluet, et al. "Querying and Updating the File" *Proc. VLDB*, 1993, pp. 73-84.
- [13] A. Deutsch, M. F. Fernandez, et al. "Storing Semi Structured Data with STORED", *Proc. ACM SIGMOD*, 1999, pp. 431-442.
- [14] D. Florescu and D. Kossman, "Storing and Querying XML Data using an RDMBS", *IEEE Data Engineering Bulletin* 22(3), 1999, pp. 27-34.
- [15] J. Shanmugasundaram, K. Tufte, et al. "Relational Databases for Querying XML Documents: Limitations and Opportunities", *Proc. VLDB*, 1999, pp. 302-214.
- [16] Andrew Eisenberg and Jim Melton, "SQL:1999, formerly known as SQL3", *ACM SIGMOD Record*, 28(1), Mar 1999, pp. 131-138.
- [17] S. Abiteboul, et al. "Data on the Web", Morgan Kaufmann Publishers, 2000.

- [18] S. Amer-Yahia and M. Fernandez, “Overview of Existing XML Storage Techniques”. Available at <http://www.research.att.com/~sihem/publications/SIGRECORD02.pdf>
- [19] F. Tian, et al., “The Design and Performance Evaluation of Alternative XML Storage Strategies”, Technical report, University of Wisconsin, 2000. Available at <http://www.cs.wisc.edu/niagara/papers/vldb00XML.pdf>
- [20] W3C Document Object Model(DOM) Level 1 Specification, available at <http://www.w3.org/TR/REC-DOM-Level-1>
- [21] W3C Document Object Model(DOM) Level 2 Core Specification, available at <http://www.w3.org/TR/DOM-Level-2-Core>
- [22] W3C Document Object Model(DOM) Level 3 Core Specification, available at <http://www.w3.org/TR/2004/PR-DOM-Level-3-Core-20040205>
- [23] XQuery 1.0 and XPath 2.0 Data Model (W3C Working Draft 12 November 2003), available at <http://www.w3.org/TR/xpath-datamodel>
- [24] XML Schema Part 2: Datatypes, available at <http://www.w3.org/TR/xmlschema-2>
- [25] XML Path Language (XPath) 2.0, available at <http://www.w3.org/TR/2003/WD-xpath20-20031112>
- [26] XQuery 1.0: An XML Query Language, available at <http://www.w3.org/TR/xquery>
- [27] Oracle XML DB White Paper, available at <http://otn.oracle.com/tech/xml/xmldb/Current/TWP.pdf>

- [28] Oracle XML DB Demo, available at
<http://otn.oracle.com/tech/xml/xmlldb/9.2.0.2.0/basicDemo.pdf>
- [29] eXtc: XML Technology for Caché Overview Document, available at
http://www.mgateway.tzo.com/extc0verview3_0.pdf
- [30] Caché On-Line Documentation, available at
<http://platinum.intersystems.com/csp/docbook/DocBook.UI.Page.cls>
- [31] Berkeley DB XML: <http://www.sleepycat.com/products/xml.shtml>
- [32] Berkeley DB XML Product Data Sheet, available at
http://www.sleepycat.com/products/pdfs/ds_dbxml1.pdf
- [33] Berkeley DB XML: Getting Started with DB XML for C++, available at
http://www.sleepycat.com/xmldocs/pdf_xml/GettingStartedCXX.pdf
- [34] eXist: <http://exist.sourceforge.net/index.html>
- [35] eXist On-line Documentation: <http://exist-db.org/performance.html>
- [36] Apache Xindice: <http://xml.apache.org/xindice/guide-administrator.html>