Towards An Enterprise XML Architecture

Ravi Murthy, Zhen Hua Liu, Muralidhar Krishnaprasad, Sivasankaran Chandrasekar, Anh-Tuan Tran,
Eric Sedlar, Daniela Florescu, Susan Kotsovolos, Nipun Agarwal, Vikas Arora, Viswanathan Krishnamurthy
Oracle Corporation
firstname.lastname@oracle.com

ABSTRACT

XML is being increasingly used in diverse domains ranging from data and application integration to content management. Oracle provides an enterprise wide platform for managing all types of XML content. Within the Oracle database and the application server, the XML content can be efficiently stored using a variety of storage and indexing methods and it can be processed using multiple standard languages within different programmatic environments.

1. Introduction

XML has gained widespread use across the enterprise in domains such as application integration and content management. The XML content from these domains covers the full spectrum from totally structured information, through semi-structured to purely unstructured text. Oracle provides a broad set of solutions for the efficient management of the diverse types of XML content within the database and application servers.

The Oracle database and the application servers [1] have a rich set of capabilities for efficient XML data storage and processing and it implements several standards for XML generation, validation, transformation, query and update operations. The paper describes the set of alternatives for storing and processing XML data and the global architecture of the XML solution available within the Oracle products.

The rest of the paper is organized as follows. Section 2 describes the XML data representation, storage and indexing methods. Section 3 discusses general techniques in XML processing including XML generation, schema validation, XSLT transformation, XPath query and XML updates. Section 4 provides a detailed look at the XQuery support in the database and application server tiers. Section 5 concludes.

2. XML Data Representation

All the XML data processing languages and environments available on the Oracle platform rely on *XMLType* - a fundamental abstract data type supported across all tiers, including the database and application servers. XMLType can be used as a base type for columns in tables or views and as a parameter type for stored procedures. Its underlying logical data model is the XQuery Data Model [4]. An instance of this data model is a sequence of items, where an item can be either a node or an atomic value. Oracle system treats the XMLType datatype as a logical abstraction over a variety of physical storage forms.

The choice of the particular storage characteristics is either decided by the user or automatically by the system. The role of the XMLType is exemplified in the figure 1.

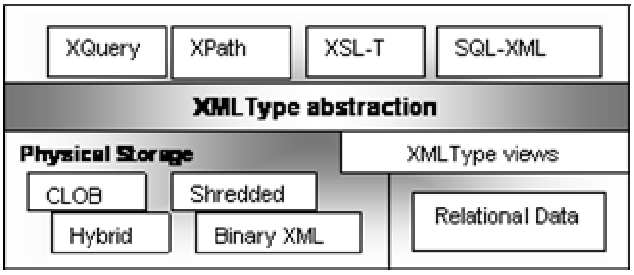

Figure 1. The role of the XMLType abstract data type

2.1. Many Flavors of Storage

Since XML is used in many diverse scenarios ranging from data interchange to content management, it becomes obvious that there is no single storage strategy that is optimized for all use cases. To satisfy the broad range of requirements, Oracle provides a spectrum of storage choices. Each of the storage alternatives offers unique tradeoffs that make it optimal for certain usage scenarios, and suboptimal for other usage scenarios. However, it should be noted that the choice of storage is kept completely transparent to the application developers using the XMLType; changing the underlying storage format does not affect the application code, akin to relational views.

2.1.1 CLOB and Shredded Storage

At one end of the storage spectrum, Oracle supports storing XML data as CLOBs. The *CLOB* storage provides document (or textual) fidelity, as the original bytes are preserved. This alternative offers the best data ingestion and retrieval performance but typically exhibits query and update performance limitations. Consequently, this storage is optimal for the scenarios where entire documents need to be stored and retrieved as a whole, with little or no further processing.

At the other end of the spectrum, if the XML data has an associated XML schema, then the data may be stored as fully shredded into object-relational rows. The shredded storage usually offers good query and value replacement performance but incurs overheads during node insertions or deletions. This kind of storage is very important in scenarios where the XML data must coexist with relational data. By shredding the XML data into tables, the data becomes available directly to the SQL engine, allowing it to perform global optimizations across a SQL statement accessing relational and XML content together.

Oracle XML DB also supports a hybrid storage format in the spectrum that ranges between CLOB and full shredding. The XML Schema that accompanies a certain XML document can be

SIGMOD 2005, June 14-16, 2005, Baltimore, Maryland, USA

annotated. The annotations mark certain portions of the data as shredded, and others being stored as CLOBs. This hybrid scheme offers the ability to optimize and adapt the storage based on application query and update requirements.

2.1.2 Binary XML Storage

There are many scenarios where the shredded storage is not a feasible option. In some cases, the XML schemas are very large and complex, resulting in thousands of object-relational tables. Often the data is also very sparse leading to a large number of null values. There are obvious performance and manageability issues with such schemas.

Oracle has developed a *binary XML format* and storage scheme, which is designed to combine the benefits of shredding and CLOB options. The binary XML format is based on a tag tokenization, hence eliminating the overhead of start and end tags and leading to good data compression. In the presence of a schema, there are further format optimizations. The data is encoded in its native typed format such as integers and floats. Other schema constraints such as ordering and cardinality are exploited for further optimizations.

The ability to break up XML tress into disjoint fragments is natively supported in the format via the notion of section references. This allows the storage system to fragment the XML trees at arbitrary boundaries and recompose them efficiently.

The XML data represented in a binary format can be stored in one or more XMLType table containers (in underlying BLOB columns). The key difference from shredded storage is that the table schema is decoupled from the XML schema. Instead of a one-to-one correspondence between XML schema elements and table/columns, there is a many-to-one mapping of XML tree fragments to binary XML tables. The correspondence can be explicitly defined by the user or implicitly chosen by the system. The binary XML storage system is expected to be available in a future release of Oracle.

The binary XML format is also supported in all tiers of the Oracle enterprise stack including the Oracle application server and Oracle Web Cache leading to improved throughput and efficiency when XML data is transferred across multiple components, processors and tiers. It should be noted that a W3C working group activity is currently underway to investigate the requirements to standardize a binary format for XML [9].

2.2 XML Indexes

Oracle provides a choice of indexing techniques that are tailored to accelerate operations over the different forms of storage. If the XML data is stored in shredded form, the entire relational indexing technology becomes directly applicable for XML operations. Relational-style B-Tree and bitmap indexes can be created on the underlying relational columns.

To improve the performance of operations on LOB storage, Oracle plans to introduce in a future release a new index kind called *XMLIndex*. This logical index consists of a physical path table containing rows corresponding to nodes (elements and attributes) in indexed documents. Each row of the path table consists of a document identifier, a Dewey style order key that represents the hierarchical position of the node within the document, a path identifier corresponding to the named path from the root to the node, along with the node value in case of simple elements and attributes. Secondary indexes are created on the path table to provide quick access to rows based on path, position or value predicates.

A user XML query is rewritten to use the XML index path table. This rewriting step decomposes the original XPath or XQuery expression into a set of simpler navigational paths. Each such simple path component results in an access to the path table. The components are combined by means of self-joins of the path table. The self-joins are hierarchical and involve either a parent-child relationship or an ancestor-descendant relationship – both accomplished by exploiting the order key. The rewritten query is optimized by the relational optimizer which picks the appropriate join order and access methods.

2.3 XML Repository

Many content-oriented applications prefer to deal with XML documents as files in a file system rather than rows of a table. For instance, the file system abstraction is very useful to a content creator working with an XML editor to fetch, edit and save XML content. Oracle XML DB provides a file system abstraction for the XML content stored in the database. This XML repository allows XML documents stored as rows of a table to be viewed and operated as files in a file system. The repository can be accessed via protocols such as FTP and HTTP/WebDAV [10]. A new hierarchical index enables efficient path resolution. The WebDAV Access Control List (ACL) standard [11] is implemented for file level security (analogous to row level security). An extensible metadata model using XML is provided for associating and managing application defined metadata for XML files. Further, traditional content management features such as versioning are also natively available.

3. XML Processing: Basic Techniques

XML data can be accessed and processed using a variety of languages within a wide range of environments. Specifically, it can be queried, updated or transformed using SQL/XML, XPath 1.0 or 2.0, XQuery and XSLT. They are simply considered as alternative language syntaxes. Programs in these languages are processed similarly: they are compiled into the same underlying algebraic representation and are further optimized based on the underlying physical characteristics of the XML data. The support for different languages and environments is discussed in sections 3 and 4.

3.1 XML Publishing

The vast amounts of data stored in the relational form are a natural and important source of XML data. Since the release of Oracle Version 8i, the Oracle RDBMS has been fully extended to an object-relational architecture that allows for the storage and processing of object types, collection types, and opaque types inside the database kernel. Both relational and object-relational data can be converted into XML through various mechanisms.

The most straightforward way of converting the result of an arbitrary relational query to XML is to define a standard mapping from relational rows and columns into XML elements and sub-elements/attributes with various options of how to map NULL values etc. This has been standardized in SQL/XML 2003 [8]. Oracle provides the DBMS_XMLGEN PL/SQL package to support generation of XML from both relational and object relational queries with various standard mapping options and extensions.

XML is hierarchical in nature while the relational data is flat. Therefore, the more useful kind of XML generation from relational data sources requires mechanisms to generate arbitrary shaped XML trees from relational tables. This is accomplished using the XML publishing functions, such as XMLElement(), XMLForest(), XMLAgg(), XMLConcat(), as defined in SQL/XML 2003 standard [8]. These new SQL functions enable users to construct arbitrary shaped XML data. In particular, XMLAgg() used with nested correlated sub queries can generate nested master-detail hierarchical XML.

Unlike the processing of classical relational functions returning small size scalar values with bottom-up evaluation strategy, the XML generation benefits from top-down streaming evaluation of SQL/XML publishing functions and infrastructure of supporting large size CLOB/BLOB data.

3.2 XML Views

Using the SQL/XML publishing functions it is possible to create virtual XML views over object-relational tables and views without physically migrating the data into the XML form. This provides a very powerful mechanism for bridging XML and object-relational data in a general enterprise architecture. It is of course true that the XML views beneficiate from the same level of query optimizations that is well known and understood for relational views.

3.3 XML Schema Validation

XML Schemas play an important role in the overall XML architecture. Both schema validated and non-validated XML data can be stored and processed within Oracle's platform [17]. XML Schemas can be optionally declared to constrain abstract XMLType tables and views. In addition, the XML Schema validation operation can be explicitly invoked via the XMLValidate() SQL function .

There are both static and dynamic implications of XML Schema declaration and validation. Schema based XMLType constraints on views over object relational data can be checked statically during view creation or compilation time, in a manner that is integrated with the XQuery static type checking. Moreover, a stream-based XML Schema validation operation has also been added to the core database server.

3.4 XPath Processing

Since the release of Oracle Version 9i, Oracle has supported querying XML using XPath expressions embedded in Extract(), ExistsNode(), ExtractValue() and XMLSequence() functions in SQL.

The XPath expressions embedded in the Extract() SQL operator serve as logical operators over XML data. They can be efficiently processed, either statically or dynamically, depending on the physical characteristic of the XML data. For the XML data whose underlying storage is shredded, the XPath operator can be compiled statically into a direct access to the underlying table columns and their SQL expressions [14, 15]. If the XML data is indexed using the XMLIndex, the XPath operator can be implemented with queries directly operating on the underlying index storage tables.

For CLOB/BLOB based XML data storage, streaming based XPath evaluation provide enhanced scalability. An XPath expression can be normalized into a path access and extra predicate evaluations. The path access can be processed as finite state automata over the XML binary token stream or XML text and the predicate evaluation can be processed as high level SQL operators. Finally, a lazy DOM is another scalability technique whereby only the current working set of the DOM objects is loaded into memory, and XPath expressions can be evaluated efficiently.

3.5 Full Text Processing

One of the important XML query use cases is the ability to merge text search on structured and unstructured content. Oracle XMLDB adds some text support in XPath by introducing the *ora:contains()* function that can execute text queries on any node. This function is optimized to use any indexes that are present on the underlying storage.

Currently the text indexes operate at the granularity of the document, i.e. each word in the inverted list contains a list of documents that they are present in. The challenge is to perform full text search at a lower granularity and retrieve only the relevant nodes in the XML tree that contain the required text. This requires a node identity mechanism that can be used in the full text indexes. The biggest challenge here is to come up with a node identification mechanism that can be used uniformly for all forms of XML data storage.

3.6 XSLT Transformation

Oracle has built general purpose XSLT 1.0 processor libraries that can be loaded into standalone XML applications or directly embedded in the Oracle database server to support the XMLTransform() SQL function. The general purpose XSLT processor is based on a virtual machine architecture. It compiles XSLT style sheets into byte code understood by the XSLT virtual machine. The byte code is cached in the database server to speed up the repeated invocations of the same XSLT style sheet over multiple XMLType instances. We are considering for the future a tighter integration between the XSLT and the XQuery processors using a common engine architecture.

3.7 Programmatic Access

In addition to declarative processing languages like XQuery or XSLT, the XML data stored in the Oracle database can also be accessed through a variety of imperative languages and APIs. In the Java environment, there is full support for XMLType in JDBC drivers. The XML data fetched from the database can be manifested into a DOM object or processed as a stream of SAX events. The interaction between the mid-tier application consuming the XML data and the database backend storing the data is highly optimized. The DOM tree constructed from the database content is lazily manifested; when the application tries to access an un-manifested DOM node, it is transparently fetched from the database. Further, when the memory usage exceeds certain limits, least recently accessed nodes are automatically freed. These optimizations greatly reduce the memory required by applications to process large XML documents.

Similar support for efficiently accessing and manipulating XML data in the C and C++ environments is provided.

3.8 XML Updates

Although an effort to provide an XML update capability is under way as part of the W3C, currently there is no such standard available. Oracle has introduced several SQL XML modification functions to address the update of XMLType in the SQL system. These SQL functions are: UpdateXML(), DeleteXML(),

InsertChildXML(), InsertXMLBefore() and AppendChildXML(). These functions take as input an XMLType value and an XPath expression. The XPath identifies the nodes to be modified and the modification is applied to the input XML data. If such SQL functions are used as L-values, then the actual modification applies to the underlying persistent XML value. Otherwise, the modified XML values are the transient return values from the SQL function and the persistent data remains unchanged. Similar to the extract() function, the XML modification functions are logical operators that can be processed efficiently depending on the physical characteristics of the input XML data. For example, if the input XML data is shredded, the DML function can be statically compiled into SQL DML operations on the underlying storage tables.

4. XQuery support

With XQuery becoming a W3C Recommendation soon, and the XMLQuery() and XMLTable() constructs being adopted in the next version of the SQL/XML standard [8], Oracle invested a lot of effort in building serious server side XQuery support. The XQuery support exploits of course the previously existing infrastructure for XML generation, XML schema validation and XPath processing that we described in the previous section.

XQuery is a very versatile language that can query and transform any data that can be abstracted via the XQuery Data Model. Since relational data, Java objects and other application data can be logically converted to XML, XQuery can be used as a uniform query and transformation language for both the backend and the mid-tier data. Supporting XQuery in only one tier would require that data in other tiers be marshaled and sent to that tier for processing. Oracle XMLDB avoids the data migration by providing a Java based engine specially designed for the middle tier and a SQL native XQuery engine in the backend database server.

4.1 Server-side XQuery Support

A lot of research addressed in the past the issue of efficient database side XQuery processing. One approach would be to add XQuery to the core database server as a standalone co-processor. Despite its simplicity, this approach has negative consequences: the database system would not be aware of the XQuery layer and consequently could not optimize queries across XQuery and relational data.

Oracle's XQuery implementation avoids this situation. The XQuery code is compiled directly into the native relational structures, thereby allowing all the standard relational optimization techniques to be applied [16]. Thus SQL and XQuery are just one of many language syntaxes that can be optimally executed by the underlying engine.

Our compilation approach compiles XQuery into the same internal data structures as SQL such as sub query blocks and SQL operators with extensions as needed. This approach enables the query to be compiled and optimized as a whole as opposed to multiple individual SQL statements.

4.1.1 Hybrid approach to XQuery processing

There are cases where the current relational system cannot be easily extended to support some of the XQuery constructs. For those cases we use a hybrid approach where the portions of the query that cannot be executed using the extended relational algebra are executed in a co-processor. The query sent to the relational optimizer is in this case a relational query with several XQuery black box operators. This approach allows us to execute the entire XQuery language in the database server, without any limitation.

The co-processor we use for this purpose is the same Java engine that executes in the middle tier. The particularity of our middle tier XQuery engine is that it is compiled into C in order to be integrated within the database server. Moreover, there are several tight integration points. For example the Java engine reuses certain components of the database engine such as the lazy DOM and the text search engine.

4.1.2 Phases of XQuery Processing

The server side XQuery processing goes through several phases.

Parsing and Semantic Analysis: The XQuery is parsed using a grammar based parser. The parser validates the syntax and generates XQueryX [5] – the XML representation of XQuery, which is then passed to the semantic analyzer.

Optimistic Static Type checking: Static type checking is very useful for early error detection and optimization. The XQuery formal semantics specification [6] defines the static type checking rules for XQuery. However, the formal semantics static typing is too strict to be useful for general purpose XQuery. We follow an optimistic static type checking approach. If the conservative type checking fails on any particular expression, we introduce dynamic type verification operations on such expressions and continue the type checking process.

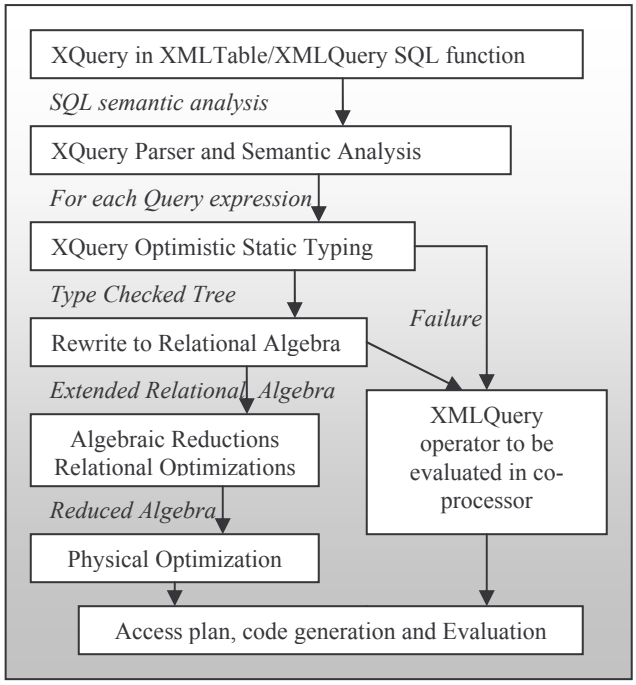

Figure 2. XQuery evaluation in the server

Translating to the extended relational algebra: The typed operator tree is then compiled into the relational algebra by converting each expression into the extended relational algebra equivalent. For example, a FLWOR expression gets transformed into a relational query block. The relational algebra has been extended to support primitive XML operations such as construction and path traversal and XQuery operations that are

not supported in the SQL engine, such as data model construction, type match, etc.

Enhancing relational query transformations: Since a lot of new XQuery operators and data types have been added to the relational system, the relational optimizer has been enhanced to handle their transformation and optimization [16]. For instance, the construction of XML nodes and path traversal to extract fragments are inverses of each other and can be cancelled out. The application of this transformation and other algebraic reduction rewritings often translates the XQuery into simple relational structures that can be easily optimized using classical relational optimization techniques.

Physical optimization: XQuery can operate upon XML data that can be stored in many forms. Thus there may exist multiple physical optimization strategies for optimizing the path expressions in XQuery. For example, the XML may be stored shredded and have an XMLIndex present. The choice of using an XMLIndex or using the shredded storage itself requires rewriting the path expressions and introducing more query blocks that need to be further optimized.

This complexity is addressed in Oracle XMLDB by compiling the XQuery into a canonical relational algebra with primitive operators for path traversal. For instance, XPath expressions containing predicates are normalized into query blocks containing simple path traversals and where clauses containing the predicates. Then, a specific storage-aware optimization strategy for each of the path traversal is chosen based on either a rule or the estimated cost and the query is further rewritten using that strategy.

4.2 Application and client tier XQuery support

Since a significant amount of XML processing occurs in the application tier or the client tier, Oracle also provides a 100% Java based engine that can be used in those tiers for querying XML, similar in spirit to other middle tier XQuery processors. The Java engine can run standalone on the file-system and is also integrated with the Oracle J2EE application server.

4.2.1 XML Query Service (XQS)

Oracle provides an XML–based data integration component called XML Query Service (XQS). XQS executes general XML queries over any data source in the application server. Various adapters are provided over the input sources to virtually and physically convert non-XML data into XML form. Oracle XQS also provides caching facilities to cache the sources and results of XQuery evaluation. Hence, a declarative querying capability is available not only for the backend database server, but for all XML and non-XML data in the application tier. At the end of the processing the result of the query can be converted into Java objects using JAXB [12] or using Oracle TopLink [2].

4.2.2 XQJ API for querying XML

The Java Community is developing a new Java API for XQuery, called XQJ – XQuery API for Java, through JSR 225[13]. Since XQuery can be used in any of the tiers, this API is being built to be agnostic of where the actual processing actually occurs. Oracle plans to provide seamless XQJ access to both mid-tier Java based XQuery engine and a backend native XQuery processor.

4.3 Query and Data Migration across Tiers

When evaluating XQuery in the middle tier, the data may be coming from the backend or from the middle tier. If the data is backend resident, it might be more efficient to send (parts of) the query to the backend for processing where it may be further optimized to use existing indexes. A rule-based strategy is used today to detect the biggest possible sub-expression of the given XQuery to be sent to the backend. In a different context, the XQuery in the middle tier may be operating upon XQuery views in the backend and it may be more efficient to retrieve the view definition and optimize it as a single query in the middle tier. Such optimizations have been made possible because the query language and the data model are the same across all tiers.

Oracle XMLDB is uniquely positioned to capitalize on the data or query migration opportunities since it has query engines that run on all tiers which can handle the same logical and physical XQuery data model.

5. Conclusions

This paper described the Oracle enterprise-wide platform for managing XML content. The platform, consisting of the Oracle database and application servers is based on a large number of current and upcoming industry standards, and provides flexible and efficient XML processing.

6. REFERENCES

[1] Oracle Application Server 10g. http://www.oracle.com/technology/products/ias/index.html

[2] Oracle TopLink - Java Object to relational persistence mapper. http://www.oracle.com/technology/products/ias/toplink/index.html

[3] XQuery 1.0: An XML Query Language http://www.w3c.org/TR/xquery

[4] XQuery 1.0 and Xpath 2.0 Data Model http://www.w3c.org/TR/xpath-datamodel

[5] XML Syntax for XQuery 1.0 (XQueryX) http://www.w3.org/TR/2003/WD-xqueryx-20031219/

[6] XQuery 10.0 and Xpath 2.0 Formal Semantics http://www.w3c.org/TR/xquery-semantics

[7] The international Committee for Information Technology Standard H2.3 Task Group, http://www.sqlx.org

[8] SQL/XML 2003: ISO/IEC 9075-14:2003.

[9] W3C Binary XML Characterization Working Group. http://www.w3.org/XML/Binary/

[10] HTTP Extensions for Distributed Authoring – WebDAV. http://www.webdav.org/specs/rfc2518.html

[11] WebDAV Access Control Protocol. http://www.webdav.org/specs/rfc3744.html

[12] JAXB - Java Architecture for XML Binding - http://java.sun.com/xml/jaxb

[13] JSR 225: XQuery API for Java (XQJ)

[14] M. Krishnaprasad, Z. Hua Liu, A. Manikutty, J. Warner, V. Arora, S. Kotsovolos. Query rewrite for XML in Oracle XMLDB. Int. Conf. on VLDB, 2004

[15] M. Krishnaprasad et al. Towards an industrial strength SQL/XML infrastructure. Int. Conf. on ICDE 2005

[16] Z. Liu, M. Krishnaprasad, V. Arora. "Native XQuery processing Oracle XMLDB", Int. Conf. Sigmod, 2005

[17] R. Murthy, S. Banerjee. XML Schemas in Oracle XML DB. Int. Conf. on VLDB, 2003