Architectural Approaches of XDBMS Realization

Muhammad Mainul Hossain

University of Kaiserslautern, AG DBIS

Abstract. Since the introduction of XML in the late 90's, emerging number of documents written in this format are being generated. A high demand for efficient storing, indexing and querying for such documents is evolving. In recent years, several efforts have been made in both research and industry areas to cope with this demand. We investigated the architectural aspects of some well-known efforts for the realization of storing and querying techniques of XML documents.

1 Introduction

Relational database management systems (RDBMSs) have enjoyed a widespread success in the last two decades. Huge number of these systems is already deployed and the data stored on them may exist many years to come. Numerous implementation techniques and efficient algorithms have been introduced successfully for these systems. On the other hand, thousands of XML documents are being generated currently every day causing strong demand for efficient storage and retrieval of such documents. An evolutionary rather than revolutionary approach would guarantee hybrid storage of relational and XML data. Various efforts have been made to reuse the current RDBMS techniques for storing, indexing and querying XML documents by mapping the data to the relational tables.

The extreme variability in size and unpredictability of use and update frequency of XML documents may reduce performance substantially and could cause huge resource overhead. Traversing a gigabyte document to retrieve a small sub-tree or rewrite it to the disk whenever a single byte is changed is expensive [2]. To store XML documents effectively and to support efficient retrieval, modification and update of these documents or parts of them, developing a native XML base management system (XDBMS) is necessary.

Traditional RDBMSs are designed with the so-called layered architecture [11] and queried with SQL language. The relational data is structured in nature. On the contrary, XML documents contain both structured and semi-structured data. There are several standardized interfaces and languages for the navigational and declarative access to these documents – DOM [13], SAX [29], XQuery [3], and XPath [1]. This work investigates architectural aspects of several well-known XML database systems, including research projects and commercial products and is structured as follows. In Section 2, we explain various possible strategies of XDBMS implementation that are observed in the literature. In Section 3, we give an overview of an "ideal" XDBMS, following [11]. Subsequent sections, Section 4 to Section 13, explain various

approaches which are currently implemented or in progress for XDBMS systems. We focus on the storage model, index structures, data access system, transaction management and concurrency control of these systems. In Section 14, we make a high-level comparison on all approaches, trying to summarize the main issues of each one and showing similarities and differences. Section 15 concludes this work.

2 Implementation Strategies

Five possible implementation strategies have been observed in literature and in commercial implementations for storing and querying XML documents. None of them is superior to the others. Each one has positive and negative points.

First, XML over Relational (**XOR**) shreds the XML documents into atomic values and stores them in individual columns of relational tables. Existing RDBMS systems components can be reused without modifications (or more realistic, with minor modifications) in this approach. But translating powerful XQuery queries into SQL queries is extremely complex and in many cases very inefficient [22]. Moreover, normalizing XML documents into relational tables as columns may substantially increase storage overhead.

Second, XML documents are stored as B**LOB** or CLOB columns into the RDBMS. The BLOB value is queried through an XQuery processor. The entire XML document must be brought into memory even to read a simple attribute before processing which may create a huge memory overhead.

Third, in **Side-By-Side** approach query fragments are translated and exchanged between XQuery and SQL processors while XML documents are stored side-by-side with relational data. This approach is more efficient than the previous two with more efficient query translation and increased degrees of freedom in the evaluation [2]. But it requires definitions of equivalent components in both worlds: Relational (SQL) and XML (XQuery) for the efficient combination and correlation of query fragments which can be extremely complex.

Fourth, in Relational over XML (**ROX**) approach the XML documents are put into a native store and an XQuery engine is built on top of it. The SQL requests are first translated into equivalent XQuery request and query on data is performed through XQuery Engine. The main challenge in this approach is the equivalent SQL-to-XQuery translation due to the semantic differences of these languages.

The last approach is a ***purely native XDBMS*** system implementation. These systems support only XQuery and do not support SQL or relational storage. So no query translation between SQL and XQuery is required and the storage and query system can be optimized for XML documents. Timber and XTC, described in section 5 and 7 respectively, are two examples of such systems.

3 Architecture of an ideal XDBMS

An “ideal” XDBMS should support all proven-effective database properties of RDBMS. The ACID properties, scalability, availability, usability, concurrency must

be supported. The system should index XML data for efficient node traversal and additionally for full-text search capabilities. Query compilation should generate optimized execution plans.

It should employ the famous layered architecture [11] which is widely applied in realization of RDBMS. The basic idea behind this architecture is creating a dynamic abstraction from the level of physical storage up to the user interface in five steps, as seen in Table 1.

Table 1. DBMS mapping hierarchy of layered architecture

	Layer	**Abstraction**	**Objects**
Data system	L5	Nonprocedural Access	Nodes, Documents, views
	L4	Navigational Access	Sequences, (logical) Records, hierarchies, networks
Access system	L3	Access Path	Physical records, access path
Storage system	L2	Propagation Control	Segments, pages
	L1	File Management	Files, blocks

Abstraction begins in layer 1 (L1) on the non-volatile storage devices and stepwise moves upward providing cleaner objects and allowing more powerful operations. During runtime these five layers can be optimized into three layers – storage system, access system, and data system. Haustein & Härder [10] proposed an architecture for native XDBMS based on the layer model which we will cover in section 5. In next sections, we detail each layer.

2.1 Storage System

Layers 1 and 2 at the bottom are responsible for the management of external storage devices and DB buffers. The repository must support rollback and recovery. Efficient retrieval and update of XML fragments must be guaranteed. Small documents could be stored completely in a single page while large documents are stored in the consecutive pages to exploit the sequential locality. Efficient buffer management with page replacement algorithms tailored to specific workload is necessary for optimization. Furthermore, related documents can be clustered together to exploit the larger pre-fetching chunks that relatively slower disk arms requirements, or possibly de-cluster them to spread across multiple arms for greater I/O parallelism [2].

2.2 Access System

To guarantee scalability in concurrent operations, XML documents must be stored in such a way that fine-granular management and locking is facilitated. The access system must enable arbitrary insertions and removals of XML fragments at any time. Various node labeling schemes have been proposed for the access model [20, 12].

Two classes of schemes are mainly found in the literature: ***range-based*** and ***prefix based***. In **range-based** schemes, positions of nodes are marked by (*DocNo, LeftPos:RightPos, LevelNo*) where *LeftPos* and *RightPos* indicate labeling range in each node with its sub-tree, generated by a depth-first traversal of the tree. If *RightPos* of a node exceeds the *LeftPos* of its sibling due to insertions, the whole tree must be relabeled. Even leaving large gaps in the numbering range, the immutability of node labels cannot be guaranteed in the general case. **Prefix-based** schemes directly encode the parent of a node as a prefix of its label. The node labels are immutable but in deep trees these labels length may grow largely requiring efficient compression techniques on these schemes.

Indexing for node traversal and full-text search must be optimized to quickly locate the intended elements. Similar to relational counterpart variants of B*-trees can be applied here. The integrity constraints and relationships among multiple documents must be realized here. XML data has either highly dynamic schema which changes frequently or no schema at all. A schema-less approach during document storage and application of schema during data retrieval may simplify querying of data. A schema-aware approach is useful to type inference and some situations in query optimization. A strict schema-based approach may restrict the flexibility of XML whereas allow to explore more storage and query optimization opportunities. Nevertheless, the issue on what would be a real XML database schema, and what kind of benefits could be derived from, is open so far.

2.3 Data System

The data system consisting of uppermost two layers provides navigational access to the internal representation of physical records and declarative operations to the XDBMS. Query operations are targeted to some logical object representation independent of the access path. Various scan operations can be implemented for record-at-a-time processing of physical record set which in turn supports sequence-at-a-time processing of XQuery and XPath. Various plan operators must be implemented for selection, join, projection and update in a single document or across multiple documents.

2.4 Transactions

The support of effective transaction is mandatory in databases. Adoption of transaction related activation and surveillance tasks must be implemented in the layer 4. For performance reasons, layer crossing information is allowed for transaction. At storage system, effective concurrency control and efficient logging/recovery techniques can be applied to achieve atomicity [8]. Collaborative use of XML documents in multi-user environment is guaranteed by fine-granular concurrency control and efficient lock management must be implemented to achieve it [12, 10].

4 ROX

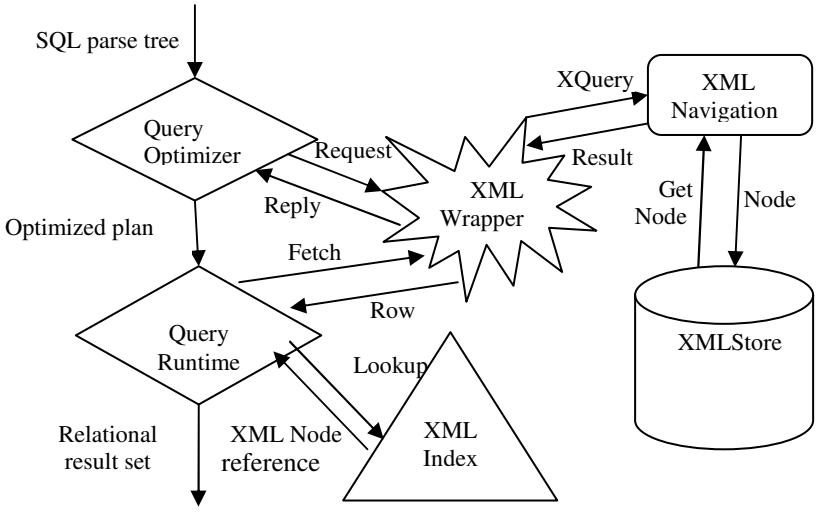

Fig. 1. ROX Architecture

ROX [9] is an experimental model to query XML documents through both SQL and XQuery. It proposes storing XML documents natively and building up an XQuery engine to query them. SQL is implemented on top of this engine providing SQL-to-XQuery translation.

The key to this coexistence of relational and XML data is the degree of translation between SQL and XQuery languages. The SQL language is defined over the relational model [32], where queries operate over column values. On the contrary, XQuery [3] is defined over XQuery Data Model (XDM) [5] which manipulates ordered, heterogeneous sequences of values and node references. XDM is more powerful and elaborated than its relational counterpart. So XQuery-to-SQL translation can only represent a subset of this data model. Along with the differences in data model, there exist also differences in operational semantics such as the document order preserving semantics of XQuery.

Despite the different focus of each language, they both include many similar concepts including set-based and sequence-based processing, joins, selections, projections, and quantification [2]. XQuery is a powerful functional language designed to query both structured and unstructured data. Therefore, there are considerable conceptual overlaps in the functionality of SQL and XQuery. When constrained over structured data, many XQuery operations have semantics close to that of SQL [2]. Many SQL predicates can effortlessly be translated into equivalent XQuery predicates. So some architecture like ROX considers SQL-to-XQuery as a viable candidate for running SQL queries over XQuery engine.

To avoid huge task of implementing the complex ROX architecture, a prototype infrastructure on top of some existing products has been proposed in [9] as depicted in Fig 1. For the native storage of XML documents a prototype XML store is developed.

The data are first parsed and stored in this store as a native tree format. A modified version of XML Wrapper, part of IBM DB2 Information Integrator [23], is used as an interface to this store.

XML Wrapper parses XML documents stored as text files on Disk with Xerces [33] XML parser and query data with Xalan XPath evaluator [34]. In DB2, CREATE NICKNAME statement queries XML documents and creates relational rows using this product. To comply with the parent-child relationship, XML wrapper allows specific columns to be specified as the PRIMARY_KEY or FOREIGN_KEY for a nickname which can be used for SQL joins.

The actual XML Wrapper is modified such that it works on the XML store rather than on the XML disk files. Xerces parser and Xalan XPath evaluator are no more needed. The PRIMARY_KEY option is implemented through an internally generated XML node identifier.

To run SQL queries over an XQuery implementation it is necessary to translate SQL queries to XQuery. Due to the semantic differences of these languages, special care must be taken during translation. SQL value comparison operators have similar semantics as those of XQuery. Furthermore, many SQL predicates over numeric types and strings can be translated into XQuery functions. The authors of [2] are convinced that with minor implementation efforts a large set of SQL predicates over numeric types and strings can be translated into equivalent XQuery predicates.

The native storage of XML documents permits both normalization and denormalization of XML documents. So the decision whether to normalize or denormalize the data is left to the database designer. The techniques of materialized views applied in the relational world to speed up evaluation of queries can be reused here. XML indexing is proposed to find documents that contain a certain value in a certain location which facilitates efficient value-based and selection joins.

5 Timber

Timer [15] is a native XML database built on top of Shore [31], a popular storage manager. For the efficient processing on large databases, a bulk (set-oriented) algebra, called TAX, has been developed which provides set-a-time processing of trees. Timber architecture is shown Fig. 2.

5.1 XML storage

The XML document enters into the system through the Data Parser which converts it into a parse tree. The data manager transforms each node of the parse tree into an internal representation and stores into Shore. There is a node for each element whose attributes are stored into a single child node, and the content into another child node. If the node is of a mixed type, each content part is pulled into a separate child node. Current Timber version ignores all processing instructions, comments etc. It employs a node labeling scheme with *start*, *end*, and *level* labels (a variant of the range-based labeling). It is possible to assign a *doc* label if document boundary is important for a multi-document model. The same can be achieved in a single model by using a

predefined range of label values for each document. Elements and sub-elements are clustered together. The XML documents are normally stored without any schema. But a *metadata* store is developed to store information regarding attribute types, data set sizes and indices if schema is available.

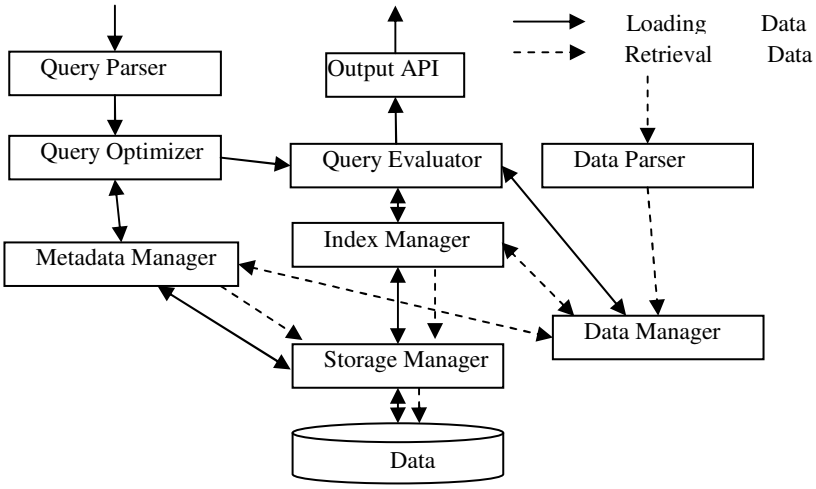

Fig. 2. Timber Architecture

5.2 Indexing

Timber uses B-Tree index facility provided by Shore to store indexes. Value indices are constructed on attribute values and element content, term-based inverted indices are built on element content for large text, and index on tag name is implemented to return all elements with the specified tag. Index returns list of *start*, *end* and *level* labels. If *metadata* store contains schema information of a document, *index manager* uses the information to select appropriate index structures and optimize query.

5.3 Query

The *Query Parser* parses XQuery into an algebraic operator tree made from **TAX** algebra operators. The Query Optimizer reorganizes this tree and maps from logical to physical operators creating a query plan which is evaluated by the Query Evaluator. Each operator of the bulk algebra, **TAX**, takes one or more sets of trees as input and produces a set of trees as output. It supports operators for *selection*, *projection*, *reordering*, *grouping*, *product*, *set union*, *set difference*, and *renaming* [16].

A tuple in relational algebra can be uniquely referenced by the attribute name or position. The same referencing is more difficult for trees due to their complex structures. TAX uses *pattern trees* to specify homogeneous tuples of node bindings. It is possible to find the matching sub-trees using this pattern. The return tree is also

homogeneous and is called witness tree, since it bears witness to the success of the pattern match on the input tree of interest. Pattern trees can hold various conditions and simplify XPath query evaluation.

6 Natix

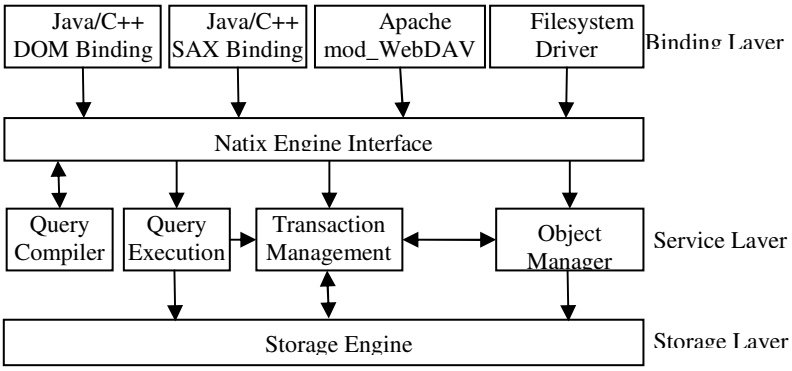

Fig. 3. Natix Engine Architecture

Natix [5] provides native storage of XML and support of XQuery, XPath, SAX and DOM interfaces. The native architecture is based on three layers: storage layer, service layer and binding layer as shown in Fig. 3.

6.1 Storage layer

Storage layer provides an abstraction from the block devices to manage the persistent data structures for storing XML documents and recovery log. The storage is comprised of partitions which manipulates disk pages. Disk pages are grouped into segments and their contents are accessed by page interpreters. A buffer manager synchronizes multithreaded access to the data pages and exploits referential locality to avoid expensive I/O operations.

Subtrees in Natix storage are clustered together in a record retaining the inner structure. The clustering can be tuned together to meet specific application requirements using a split matrix. The segment interface for XML allows accessing an unordered set of trees. A wrapper maps logical data model to XML model and vice versa. Nodes are labeled with a symbol taken from an alphabet Σ_{Tags}. Leaf-nodes are, in addition to this symbol, labeled with arbitrary long strings. A record is stored only in a single page. The logical data tree is partitioned into subtrees each of which is stored in a record. Large documents are split semantically into subtrees based on the tree structure and stored into several records which are linked by a special type of tree node called *proxy node*.

Natix introduced a *split* algorithm if a sub-tree exceeds page size due to update. The sub-tree of one record is partitioned into a *left partition*, a *right partition* and a *separator*. The separator is moved to the parent and uses *proxies* to point to the new records to indicate where the descendant nodes were moved due to split operator. The desired ratio of the size of left and right partitions is configurable and used by *split* algorithm to determine the separator.

6.2 Service layer

Service layer provides *Natix Engine Interface* for all DBMS services to communicate with each other. Each request is first sent to the DBMS and then forwarded to the appropriate components for processing. The service components are namely Natix query execution engine (NQE), query compiler, XPath compiler, Transaction management, object management.

6.3 Binding layer

Binding layer enables various applications to bind to the Natix Engine Interface. A request to this interface can be made using C++ data types as well as through a language independent string which is parsed by a parser and transforms into a request objects allowing a simple control interface to manipulate the Natix system.

6.4 Indexing

Natix uses inverted files for full text index and extends it for semi-structured XML data. It provides a framework for incorporating different index types through a special component called ContextDescription. A special index called eXtended Access Support Relation (XASR) is developed to preserve parent/child, ancestor/descendant, and preceding/following relationships through a variant of range based labeling scheme. The tree is traversed depth first and each node is label with a d_{min} value and d_{max} value. d_{min} is the value the node is first entered and d_{max} is the value when the node is finally left. The d_{min}, d_{max} along with the element tag and document ID are stored as an entry in XASR table which facilitates identification of a node directly.

6.5 Transaction Management

ACID properties are supported through making storage engine transaction aware and developing a special transaction management module. A tuned version of ARIES protocol [22] is used for recovery. The recovery log describes each log record with a unique log-sequence-number (LSN). Segments and page interpreters, described in section 6.1, contain logging and recovery code in their data structures. Physiological logging [8] – physical to a page, logical within a page is implemented in Page interpreters. They keep a private, subsidiary log, which can be modified, reordered before publishing to log manager. This mechanism reduces log size and increases

concurrency. During insertion operation the log entries for the subsidiary log are not explicitly stored but the data pages are used as a representation for log records before publishing them to the recovery log and thus further reduce log size. The *log manager* reads and writes log records and synchronizes concurrent access to them. The *recovery manager* performs redo, undo and checkpoint operations. The *transaction manager* provides interfaces to the application programs to mark operations for transactions.

Natix reduces log size and increases performance of logging/recovery for undo by implementing *annihilator undo*. Specific operations are marked as annihilators and undo for the subsequent operations are skipped in case of rollback avoiding the entry of nextUndoLSN chain of that transaction which will not be undone explicitly, but implicitly by the annihilator. *Selective undo* and *selective redo* further optimize logging/recovery of a data page which is updated by a single transaction through *undo*. It avoids loading and modification of pages which belong to loser transactions.

6.6 Concurrency Control

Natix uses tuned versions of multi granularity locking (MGL) [8] protocol with an arbitrary number of levels for concurrency control. Lock granularity addresses at the ***segment***, ***document***, ***sub-tree*** and ***record level***. A document can be split into records using *split matrix* and parts of document with high probability of concurrent access can be stored in different records providing configurable concurrency control.

7 XTC

XTC [10] is based on the layered architecture [11], which is successfully used in numerous relational systems. XML documents are stored natively and languages like SAX, DOM, XPath and XQuery are used to access data. The data model is based on DOM with an appropriate node labeling scheme, called ***SPLID*** – a variation of the DeweyID-based [4] labeling. The architecture of XTC is shown in Figure 4.

7.1 Storage system

Storage system consisting of L1 and L2 of the layer model hides the varying external block device properties and provides pages of fixed-length which are mapped onto physical data blocks stored on secondary storage devices (disks). Each page, in turn, is compound of variable-length records representing element/attribute nodes in a XML document. Text values (text nodes, in XTC terminology) can be stored in-lined (in the same record) with their corresponding element nodes, or in a separate record/page. Buffer management employs specific page replacement algorithms tailored to the application needs.

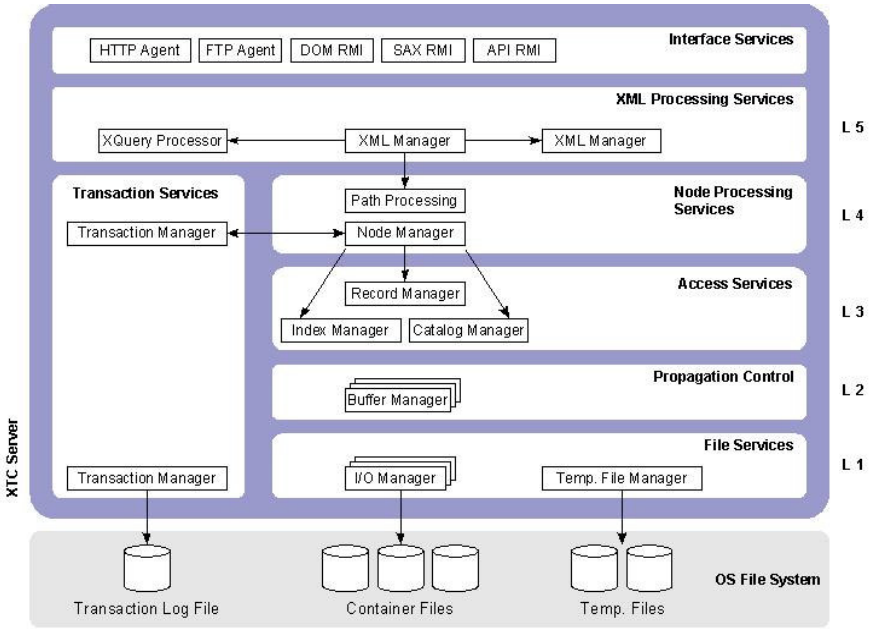

Fig. 4. XTC Architecture

7.2 Access System

The access system at L3 has been implemented using B- and B*-trees with several enhancements for single- or multi document stores, key compressions and optimal handling of short documents.

XTC employs a prefix-based labeling scheme based on the DeweyID. The scheme is flexible enough to allow arbitrary insertions and deletions without reorganizing labels already assigned. Each node label consists of divisions of integers separated by dots. The children retain the label from the parent and add one or more divisions. Initially, odd integers are used for division values and a distance is kept among the numbers to reduce overflow caused by insertions. The overflow is handled by introducing even integers. Various compression measures have been applied to reduce the storage size for long DeweyIDs. Hoffman encoding scheme is used for the distribution of distance values to reduce frequent overflows which create longer IDs.

A node together with its value part has a size limit of *max-val-size*. Any text node exceeding this limit is stored in reference mode where it is split into parts and stored into chained pages.

Indexing for documents in XTC is implemented by a document store. First node of each data page in the *document container* is indexed in a B-tree called *document index*. The document container consists of a set of chained data pages and preserves the order and cluster property of each document. All node formats are of variable length. Additionally, an element index has been developed using B*-tree to index all

elements of XML documents in a name directory. A node reference index provided for each element addresses the corresponding element using the DeweyIDs.

7.3 Data System

The data system at L4 and L5 consists of several node processing and XML processing services. The order of two nodes in the document can exclusively be determined by comparing two DeweyIDs. It facilitates the efficient evaluation of the eight axes *parent/child*, *ancestor/descendent*, *following-sibling/preceding-sibling*, *following/preceding* for declarative queries through XQuery and XPath. Furthermore, labeling scheme together with element indexes supports five basic navigational axes of DOM - *parent*, *previous-sibling*, *following-sibling*, *first-child*, and *last-child*.

Each node in the document store is mapped to a node handled by the node manager. During navigational axes evaluation, the desired node resides in the page of the context node in the best case. In the worst case, the number of pages to be accessed limits to 3-4 via *document index*. On the other hand, element index increases declarative query processing via XQuery. For query evaluation XTC uses tuned variants of structural joins and twig joins. L5 produces Query Execution Plans (QEP) i.e. translate, optimize, and bind the multi-lingual request from the navigational to the operations available at L4.

7.4 Transaction management

The *transaction manager* is responsible for providing ACID properties to XTC. Log data is collected and persisted during normal operation and used by recovery operation after system crash or media failure. Physical logging and Physiological logging combined with LSN implemented at storage layer guarantees A (atomicity) and D (durability) of ACID properties. All layers must participate to guarantee C (consistency) while L3 to L5 are responsible for achieving I (isolation). Transaction manager together with lock manager provides fine-granular locking for isolation which is described in the next section.

7.5 Concurrency Control

A lock manager, supported by 20 operations at the access model, is responsible for the collaborative processing of XML documents through fine-granular locking. It transforms the DOM tree into a so called ***taDOM*** tree introducing two new node types: *AttributeRoot* and *String Node.* *AttributeRoot* attaches all attributes of an element allowing a single lock to lock all attributes during a DOM(SAX) *getAttributes()* call. *String node* contains the value of an element/attribute node allowing avoiding lock during an existence test. Thus, with the taDOM tree, XTC enhances the transaction parallelism. Application call to access XML documents are interpreted by the lock manager to apply appropriate intention lock allowing multiple transaction to operate on single node with different lock modes.

The lock manager is realized by configurable semaphore tables which maintain the semaphore dynamically acquired for a specified maximum number of transactions and lockable objects. Several techniques have been applied to optimize performance and reduce memory requirements. Each transaction requests semaphore which performs a compatibility check through the compatibility matrix provided during semaphore table initialization. If the semaphore is compatible it is immediately set. Otherwise, it is added to the lock chain with state waiting and the requesting transaction must sleep until incompatible semaphores are removed or replaced by compatible ones. Pending transactions are reported to the transaction manager to detect and resolve deadlocks.

8 System RX

System RX [2] provides an experimental implementation of a hybrid database system of both relational and XML data (Figure 5). It proposes a native storage for efficient storing and querying of XML data. System RX uses SQL/XML language to manipulate and query XML data. SQL/XML is an industry standard language for creating and querying XML data fragments as well as relational data. It provides XML publishing functions, a new ***XML*** data type and some mapping rules between SQL and XML. The ***XML*** data type has an implementation dependent internal format different from LOB-based types. SQL/XML defines second-order query functions e.g. XMLTable, XMLQuery that take an XQuery statement as input and execute it over the XML values passed from SQL. It also provides functions to construct new XML data and convert between XML and relational data types. Several commercial e.g. Oracle [36], DB2 [26] DBMSs have already implemented SQL/XML.

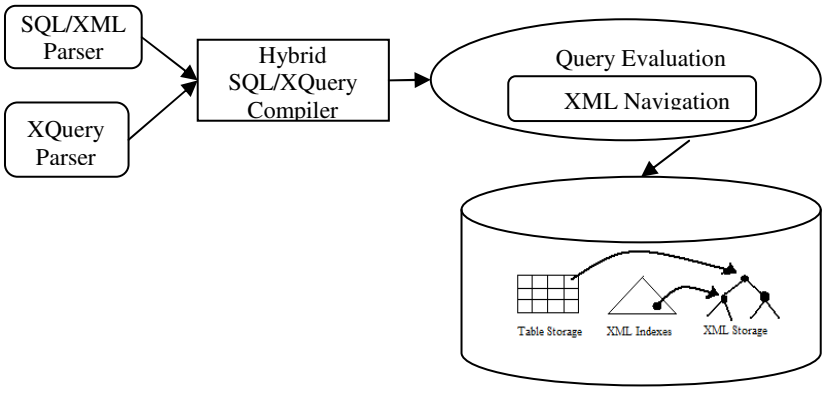

Fig. 5. System RX architecture

8.1 Storage Model

System RX stores the XML documents in the native storage as instances of XDM [5] in a structured, type annotated tree to avoid repeated parsing and validation of documents while preserving salient features like digital signatures. Node addressing and traversal are simplified through assigning unique address to each node and keeping pointers to parent and children. Related nodes are stored in standard fixed-size buffered pages. Redundant namespace URIs and node names are compressed through a dictionary which maps strings to identifiers to improve evaluation of path expressions. A set of child slots associated with each element node contains hints for fast navigation to find qualifying children without actually visiting each child node. Each child can be pointed to directly via a row identifier (RID) plus an index into child slot array. Regions of nodes are grouped together on pages and linked by a logical regions index which enables access without traversing from the root by looking up the region that contains the node in the region index using node's identifier. Nodes can be updated without affecting most of other nodes in the document. The regions that are updated are versioned, leaving multiple version of a node in the regions index. New readers will always get the latest version. Regions can be fragmented during updates. A data reorganization utility is offered to re-cluster the regions of a document.

8.2 Indexing

System RX supports three classes of XML indexes – Structural indexes, full-text indexes and value indexes with special focus on the last type. The XML index is implemented with two B+ trees. The *path index* maps each distinct reverse path to a generated path identifier. The paths are stored from leaf to root for efficient processing of descendent queries. The *value index* consists of the key (*pathId*, *value*, *nodeId*, *rid*). The *nodeId* uniquely identifies a node in a document using Dewey node identifier. The *rid* identifies a row in the table. The *pathId* allows quick retrieval of specific path queries keeping the order of the keys in the value index.

For full-text indexes, relational text indexing support [21] is extended to include XML data. To escape indexing the sheer amount of information that may arrive in XML documents; System RX supports indexing of nodes that are returned from a simple XQuery through CREATE INDEX command. Due to dynamic schema supports of XML documents, it is difficult to infer the data type associated with indexed nodes. So user must specify the data type of nodes during indexing. The index is created on the cast of the node to the indexed type, taking into consideration the node's type annotation which is derived during validation.

8.3 Query

System RX supports query compilation and evaluation for SQL/XML [14] and XQuery. Data definition (DDL) operations are performed through SQL/XML. Queries enter the system through either language and are then compiled into an

internal Query Graph Model (QGM) [27] which is then normalized, simplified and optimized through rewrite transformation. The optimizer generates a physical execution plan using this graph which is translated into executable code by code generation. The rich data-flow modeling is exploited to perform cross-language optimization. The QGM for relational system is augmented with native constructs for XML. In QGM, each path expression is represented as a pattern tree in which there is only one bound variable to represent each data flow explicitly, so that semantics and rewrite analysis, which is built on explicit data flow representation, can reason about the query more efficiently. This approach allows representing not only path expression but also other FLWOR [3] expressions and supporting full functionality of XQuery. During rewrite transformations, many techniques from the relational world are reused. It is tried to consolidate all path expression in a single FLWOR block into one pattern tree that is annotated with several flags to represent FOR vs. LET paths.

9 eXist

eXist [24] provides schema-less native storage of XML documents. XPath and XQuery languages are used to query these documents. Application binding is provided for HTTP, XML-RPC, SOAP and WebDAV. Four index files build the basic storage and indexing infrastructure for eXist: *collections.dbx, dom.dbx, elements.dbx, words.dbx*.

9.1 Storage Model

XML document nodes are stored as the root of DOM tree in *dom.dbx*. Top level elements of a document are indexed with a B+ tree which keeps a unique identifier of a node in the document to the storage address in the data pages. Access to other nodes is provided by traversing the nearest available ancestor stored in the index tree.

9.2 Indexing

The numbering scheme, based on [19], models the document tree as a k-ary tree, where k is the maximum number of child nodes of an element in the document. A unique identifier, generated through level-order traversal of the tree, is assigned to each element. To overcome the restriction on the maximum document size, the completeness constraint in the original scheme [19] is dropped in favor of recomputing the number of children that may occur at each level. All axes of XPath can be efficiently evaluated through this numbering scheme.

Collection.dbx manages the collection hierarchy which organizes the indexes for elements, attributes and keywords. *Elements.dbx* serves for the mapping of elements and attributes names to unique identifiers to locate a name in the documents. *Words.dbs* provides inverted index support which uses unique node identifier to associate words or phrases with their occurrences and locations in the documents.

9.3 Querying

The index structures along with the numbering scheme provides flexible framework for query evaluation. eXist employs an enhanced version of path join algorithm discussed in XISS system [20]. The path expression is first decomposed into several sub-expressions which are then evaluated and ancestor-descendant path join algorithm is applied on the result set. eXist does not need the actual DOM nodes in the XML except to process the equality operator which requires the values of the DOM nodes.

10 Tamino XML Server

Tamino [30] is a purely native commercial XDBMS implementation from Software AG. XML data is stored natively and queried through standard **XQuery** and XPath-based **Tamino X-Query** languages. The system consists of several components. ***X-Machine***, consisting of native ***XML Data Store*** and ***XML Engine***, is the central component of the Tamino architecture. ***XML Data Map*** component stores all schema information. ***X-Node*** connects to the external data sources. The system can be extended and customized through ***X-Tension*** framework. Fig. 6 depicts the architecture of Tamino XML Server.

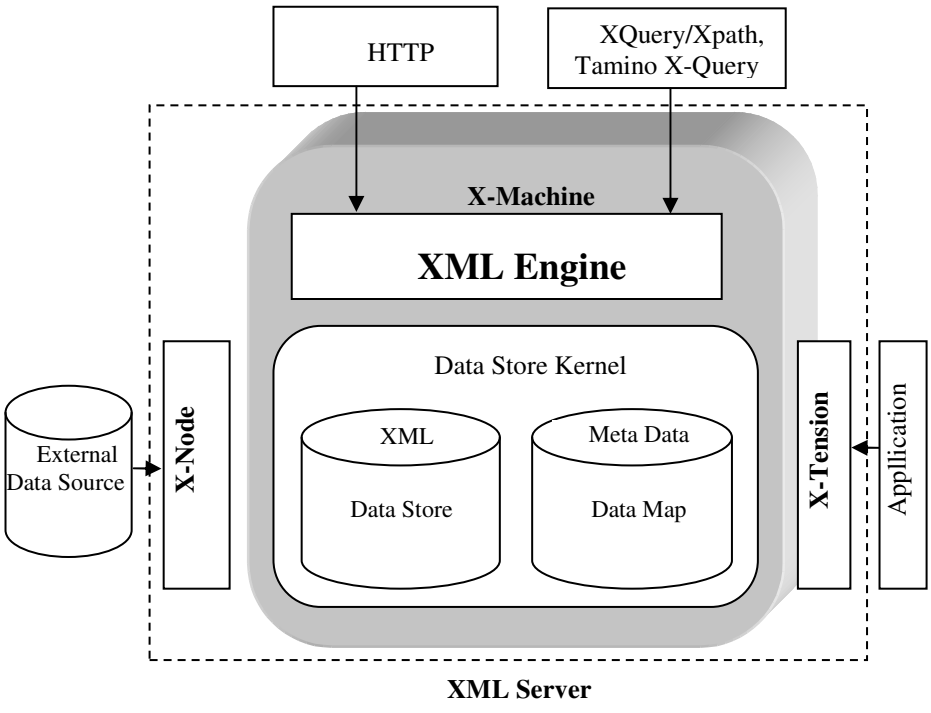

Fig. 6. Tamino XML Server Architecture

10.1 Storage Model

Incoming documents are parsed, validated against XML schemas from data map, transformed into an internal native format and stored into the XML data store. Non-XML data such as graphics and video files can also be stored. Fail-over and crash recovery are supported.

10.2 Indexing

Tamino supports three types of indexing: **value index**, **full-text index** and **structural index**. Schema information is exploited to define the scope of indexing.

10.3 Query

XML documents are queried through XQuery and Tamino X-Query, an extension to the XPath semantic to simplify multiple document querying. A well-formed XML document has only one root. If a query returns multiple XML documents (or document fragments), Tamino provides a pseudo-root element to guarantee well-formedness of the resulting XML document. Complex queries, including ***join***, involving single or across multiple documents are efficiently evaluated. Insert, update and delete operations on XML document fragments are optimized, leaving the rest of the document unchanged.

10.4 Schema support

The metadata store ***Data map*** is responsible for storing the Tamino's schemas. The schemas determine how the objects, both residing in XML documents or externally (legacy databases), can be mapped to physical structures. The schema information is used for validation during insert or update operation. Besides, Tamino schema provides a homogeneous view of external non-XML data such as legacy database as XML data to the end-user. Tamino supports both XML schema and DTD.

10.5 Transaction Management and Concurrency Control

Tamino XML server supports transaction management based on a two-phased concurrency control protocol. The data store can participate in distributed transaction scenarios while providing client-controlled, session-based support for distributed transactions. In this case, a 2PC (two-phased commit) protocol is used to guarantee transaction consistency. Tamino server establishes an indirect communication channel via a special product called Universal Transaction Platform (UTX) which communicates with transaction coordinator. Fine granular isolation is provided by hierarchical locking technique. Tamino defines four levels of locking granularities: database, collection, doctype, and document. It introduced six lock types divided into two groups: exclusive lock and intention lock. Locks in exclusive group affect the

object directly and those in intention group represent an intention to lock further object at a finer granularity. Locks are initially tried to set as fine-grained as possible, even on the node or sub-tree level. A weaker lock set can be escalated to a stronger lock. This facilitates fine-grained locking management in multi-user environment.

11 Native XML Support in DB2

DB2 Universal Database has been extended to support native XML based on System RX, described in section 8. XML and relational data are stored and queried side-by-side. SQL and XQuery can be combined and correlated during querying. XQuery, SQL/XML and SQL queries are first parsed by different parsers and processed by DB2 engine with a single compiler [26]. XML documents, defined as *XML* type, can be stored as column values along with relational data into the same table. In contrast to BLOB or CLOB type, values of type *XML* are processed in an internal native representation. XMLSERIALIZE and XMLPARSE functions can be used to convert XML to string and vice versa. Transaction management and fine-granular concurrency control are also supported for XML documents.

11.1 Storage Model

As described in the section 7, the XML data is inserted into the database as column values and stored as an XDM tree. A new catalog table SYSXMLSTRINGS is developed to map the unique tag names to distinctive StringIDs which are used to represent tags in the documents. A document tree, too large to fit on a page, is split into multiple *regions* connected by the regions index. A region index is a system index and is created for each table which contains XML columns.

11.2 Indexing

DB2 supports path-specific value indexes and full-text indexing. Usual CREATE INDEX command is used with *xmlpattern* as path for identifying XML nodes to define a value index. Xmlpattern supports a subset of XPath. Like System RX, user must explicitly specify the data type in the "*as sql <type>*" clause.

To accommodate the XML type system in the existing DB2 index manager, some values are specially handled, i.e. +0, -0, +INF (positive infinity), -INF (negative infinity) and NaN (not a number). There may be zero, one or multiple index entries for a single row that match the *xmlpattern.* In contrast, relational system returns at most one entry.

DB2's full-text search capabilities have been extended to accommodate the XML type. The documents in the XML column can be fully or partially indexed. The search can be restricted to a specific element. Standard text search features like scoring and ranking of search results as well as thesaurus-based synonym search are also provided.

11.3 XML Schema Support

XML schema validation is supported during insert, update, and query operations. Limited support for DTDs [35] is also provided. The schemas and DTDs are first registered via DB2 commands, stored procedures, or language specific APIs and stored in XML Schema Repository (XSR). The type annotation produced during validation is persisted together with the document and used at query compilation and execution. Documents can be validated in SQL statements with XMLVALIDATE function. A schema can be explicitly referenced by its schema URI or by its schema identifier. Each inserted document can potentially be validated against a different XML schema for the same column values.

DB2 supports certain degree of schema evolution in the XSR. Schema evolution is a sequence of changes in an XML schema over time. If new schema is compatible with the old schema, then the old schema can be replaced with the new schema. This only supports additions of optional elements and attributes. DB2 also allow the old and new schemas to coexist side-by-side, under different names. One can insert documents conforming to any of the schemas in the same column of a table.

DB2 provides a decomposition product to shred XML document into relational tables. It uses an annotated schema for the decomposition. The annotations enable the user to control the decomposition process such as inserting whole or part of document, defining foreign key relationship, conditional insertion etc.

12 Oracle XML DB

Oracle XML DB [36, 18] provides native storage and querying of XML documents built on a relational and object relational framework. XML data is integrated into the existing Oracle DBMS, and complete transparency and interchangeability between the XML and SQL data views are supported. XML schema registration and validation are supported as well. XML processing is based on a native *XMLType* datatype. Figure 6 shows the architecture of XML DB and following subsections detail various aspects of it.

12.1 XML Storage

Multiple storage options are available with XMLType. ***LOB-based storage*** assures complete textual fidelity ensuring high data integrity and low regeneration cost. The ***native structured XML storage*** is a shredded decomposition of XML into underlying object-relational structures for better SQL queriability. It loses whitespace information but maintain fidelity to DOM. In ***hybrid storage***, XML data is split into structured and CLOB part. The XML data can be stored in an XMLType column or using an XMLType table. Oracle **XML DB Repository**, based on WebDAV [7] standard, makes it possible to view all of XML content stored in the database using a File-Folder metaphor. It uses the WebDAV resource model to define the basic metadata that is maintained for each document stored in the repository.

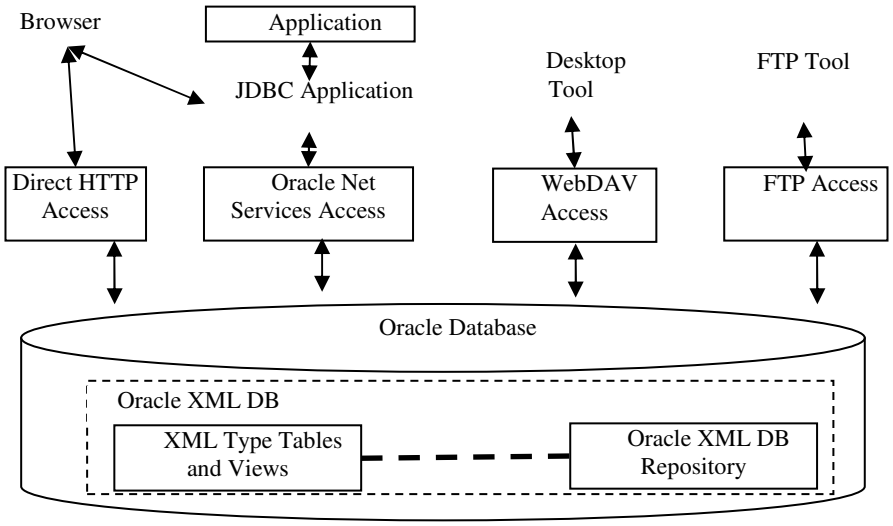

Fig. 7. Oracle XML DB Architecture

12.2 Querying

To Query XML Tables, XML DB provides several functions such as *extract()*, *existsNode()*, and *extractValue()* that use XPath to locate and extract data from XML documents. updateXML() can be used to update, replace elements, attributes and other nodes with new values. It is also possible to retrieve XML as a CLOB, VARCHAR, or NUMBER using *getClobVal()*, *getStringVal()*, or *getNumberVal()* functions respectively. When XMLType is stored in structured storage (object relational) using an XML schema and queries using XPath are used, they are rewritten to go directly to the underlying object-relational columns. This enables the use of B*Tree or other indexes to be used in query evaluation by the Optimizer.

12.3 Indexing

XMLType tables and views can be indexed using B*Tree, Oracle Text, function-based, or bitmap indexes. ***Oracle Text*** index has been extended in Oracle9i to work on XMLType columns. Operations such as CONTAINS and SCORE can be performed on XML data. CONTAINS has been enhanced with two new operators: INPATH (checks if the given word appears within the path specified) and HASPATH (checks if the given XPath is present in the XML document). Queries can be speed up by building function-based indexes on *existsNode()* or those portions of the XML document that use *extract()*. New XPath extension functions defined within the Oracle XML DB namespace, enables a richer set of text search capabilities. They can

be used within XPath queries appearing in the context of *extract()*, *existsNode()*, and *extractValue()* functions operating on XMLType instances.

13 Microsoft SQL Server 2005 XML

SQL Server 2005 integrates native XML management capabilities into the existing DBMS both extending and leveraging the relational storage and query infrastructure [28]. The XML processing is based on a native ***XML*** data type. It also provides a new metadata object called XML Schema collection to register schema and validate and enforce XML data type values with them. The architecture of SQL Server 2005 XML is depicted in Figure 7.

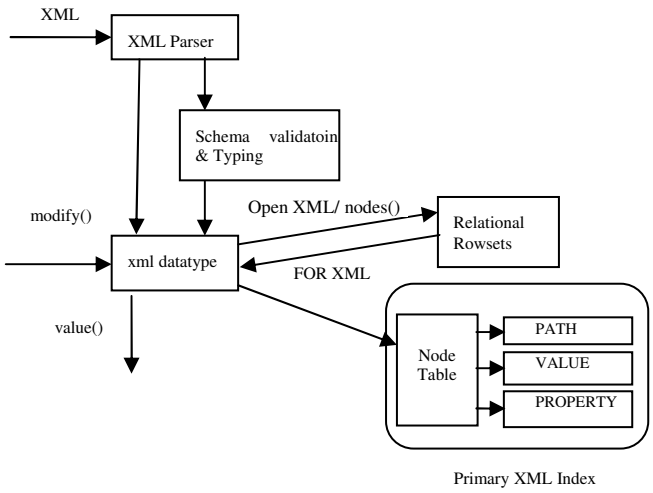

Fig. 8. SQL Server 2005 XML Architecture

13.1 Storage Model

The logical model of XML data type is based on XDM [5] which simplifies XQuery support and validation and type of data via XML schema. The physical model of XML data type is a byte level representation of the logical concepts such as element and attribute node. This internal representation can be considered as a binary encoding of XML data providing closer representation of XDM.

13.2 Querying

Querying for XML data is performed by four methods of XML data type: *query*, *path*, *exists*, *nodes*. Each of these methods takes an XQuery expression as parameter. The *query()* executes a query by evaluating an XQuery expression against the elements and attributes in an XML instance and returns an untyped XML. The *value()* is used to extract node values from an XML instance. The *exists()* method checks the existence of a specific XML fragment in an XML instance. The *nodes()* method shreds XML instance into relational data and generates a single column row per node that the XQuery expression returns. The *modify()* method xml type allows insert, update and delete content within XML instance. SQL Server 2005 introduces XML Data Modification Language (XML DML) to enhance XQuery by allowing insert, update, and delete access anywhere the xml data type is used.

13.3 Indexing

SQL Server supports two types of indexing for XML data: primary and secondary. A primary XML index is a shredded version of what is in the xml column. When this index is created, it writes several rows of data for each XML BLOB in the column. A secondary index further improves query performance on large amount of data. There are three types of secondary XML indexes: PATH, VALUE, and PROPERTY. PATH index is used to index the paths and node values as the key fields. VALUE index is useful if the queries are based on values or the path is not fully specified. The PROPERTY index is built on the key columns of the primary XML index such as Primary Key, path, or node values.

14 Comparisons

In this section, we provide a summary comparison of all approaches studied so far. Furthermore, we comment on issues, regarding comparison, of the approaches, sometimes contrasting them, sometimes pointing out similarities. We have chosen six criteria, namely, storage, indexing, query support, API support, transaction processing and XML schema support to compare the approaches. Table 2 shows the comparison information.

The major commercial databases started the support for XML data with shredding and LOB-based storage in the relational tables. Some (e.g DB2, Oracle) supported SQL/XML to query data while some (e.g. Microsoft) used OpenXML/FOR XML for that. As the databases have been enhanced with native XML processing, they continue to support former techniques to maintain backward compatibility and extend their respective relational storage engine, index facilities, transaction management and concurrency control to work with XML. Pure native approaches, on the other hand, supports only XQuery and XPath to query data. The database engine is adapted to transaction management and concurrency control targeting XML operations like *insert*, *update*, and *delete* of nodes and/or sub-trees.

Table 2. Comparisons of native XDBMS realization approaches

	Storage	Indexing	Query Support	API Support	Transaction Processing	XML Schema/DTD support
ROX	Native	Path, Value	SQL, XQuery, XPath	Not Clear	Not clear, based on underlying data engine	XML Schema
Timber	Pure Native	Value, Path, full-text, join index	XQuery, XPath	-	Based on Shore data engine	XML Schema
Natix	Pure Native	Full text, XASR (kind of path)	XQuery, XPath	SAX, DOM	Supported, extends ARIES for recovery, MGL protocol for CC	Schema-less
XTC	Pure Native	Document, Path, Element (inverted index), CAS index	XQuery, XPath	SAX, DOM	Supported, several CC protocols and taDOM3+ protocol	Schema-less
System RX	Native	Value, Structured, Full-text	XQuery, XPath, SQL, SQL/XML	SAX, DOM	Supported, use of CC protocols from underlying RDBMS	XML schema
eXist	Pure Native	Full-text, structure, value	XQuery, XPath	DOM, XML:DB	Not supported	Schema-less
DB2 UDB	Native, CLOB, Shredding	Path, Value, Full-text	XQuery, XPath, SQL, SQL/XML	DOM, SAX	Supported, use of CC protocols from underlying RDBMS	XML schema and DTD
Tamino	Pure Native	Value, Full-text, Structured	XQuery, Tamino X-Query	DOM, SAX	Supported, user of hierarchical locking for CC	XML schema and DTD
Oracle 10g	CLOB, Shredding	Function-based (bundle of path and CAS indexes), Full-text	XQuery, XPath, SQL, SQL/XML	SAX, DOM	Supported, use of CC protocols from underlying RDBMS	XML schema
SQL Server 2005 XML	Native, CLOB, Shredding	Path, Value, Property, Full-text	XQuery, XPath, SQL, OpenXML/ FOR XML	DOM	Supported, use of CC protocols from underlying RDBMS	XML schema

The focus of pure native approaches is to build up an XML document store and process queries over this store. Because of that, and to provide safe multi-user access, a hierarchical multi-granular locking mechanism is chosen by the most pure native XDBMS to deal with concurrency control (e.g., taDOM3+ protocol of XTC). In contrast, XML operations in hybrid databases (e.g., commercial databases and Timber) are generally performed over relational tables using a native ***XML* data type**. Each XML document is represented in a table column. It is not clear whether a row is completely locked (thus locking the whole document) or, in addition, a fine-grained hierarchical locking on document nodes is performed. Among all studied XDBMS, Tamino is the only one claiming to support distributed XML documents.

Range-based labeling scheme came up earlier than prefix based scheme. It is very good for query processing, but not very effective for concurrency control. Earlier approaches have mostly used range-based scheme and focused mainly on query processing (for example, Timber and Natix). Even commercial products have preferred to use range-based labeling from which we can infer that commercial XDBMS do not provide specific locking mechanism for XML data or they expect that XML documents are not volatile. The preference for prefix-based scheme is increasing as it is effective for both query processing and concurrency control. In our study, only XTC provides such a labeling mechanism.

Strong tendency to support ***path*** and ***value*** indexes has been observed. In fact almost all approaches have both indexes types, whether to accelerate the document access or to keep structural characteristics of documents. Moreover, most XDBMSs also support full-text index through inverted index structure. B- and B*-trees are the most preferred data structures for storage and indexing. Some XDBMS (e.g., eXist) focuses on quick retrieval of information and expects low rate of document modification. They use B+-trees for storage structure and indexing techniques.

Most of approaches use a schema-aware storage with XML schema or DTD. Only two of them, namely, XTC and Natix, use a schema-less storage. Some XDBMSs store XML schemas as metadata in a separate storage space and validate XML data values during load, insert and update operations. Furthermore, query processing can have benefits of schema information. For example, to select index structures and to optimize query execution plans. However, it is not clear if these stored XML schemas are used only for data type inference/validation or they are extended to capture other important information on database (e.g., number of node instances, indexes on nodes/paths, statistics, etc) and, so, can be viewed as authentic XML metadata. This separation of data and schema information enables the so-called ***dynamic schema evolution***. Using this characteristic, hybrid XDBMS, mostly commercial products (excepting Tamino), can use different XML schemas applied to a set of XML documents. Schema-less approaches need to scan XML documents to get metadata information, including data types and document structure. Moreover, statistics on document are gathered in the same way. However, as there is no validation in document load process, this metadata information may be out-of-date for query processing as document updates take place.

15 Conclusions

A lot of research efforts is being made for the optimal storage and querying of XML documents. Most of them are not yet mature enough for the enterprise applications. All major database vendors have enhanced their respective DBMS architectures to support XML processing natively. The enterprise adoption of XDBMS is increasing day-by-day. The gap of efficiency and effectiveness between proven RDBMS and emerging XDBMS tends to be narrowed in the near future.

References

1. A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, J Siméon. XML Path Language (XPath) 2.0. W3C Recommendation. 2007
2. K. Beyer, R. J. Cochrane, V. Josifovski, J. Kleewein, G. Lapis, G. Lohman, B. Lyle, F. Özcan, H. Pirahesh, N. Seemann, T. Truong, B. Van der Linden, B. Vickery, C. Zhang. System RX: One Part Relational, One Part XML. Proc. of ACM SIGMOD 2005.
3. S. Boag, D. Chamberlin, M. Fernanadez, D. Florescu, J. Robie and J. Simeon, "XQuery 1.0: An XML Query Language", February 2005, http://www.w3.org/TR/xquery
4. M. Dewey. Dewey Decimal Classification System, http://www.mtsu.edu/~vvesper/dewey.html
5. M. F. Fernandez, A. Malhotra, J. Marsh, M. Nagy and N. Walsh. XQuery 1.0 and Xpath 2.0 Data Model. http://www.w3.org/TR/xpath-datamodel/. 2007
6. T. Fiebig, S. Helmer, C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, T. Westmann: Anatomy of a native XML base management system. VLDB Journal, Vol. 11. 2002
7. Y. Goland, W. Whitehead, A. Faizi, S. Carter, D. Jensen. RFC2518: HTTP Extensions for Distributed Authoring – WEBDAV. Internet RFCs. 1999
8. J. Gray, A. Reuter. Transaction Processing: Concepts and Techniques. Morgan Kaufmann. ISBN 1-55860-190-2. 1993
9. Alan Halverson, Vanja Josifovski, Guy Lohman, Hamid Pirahesh, Mathias Mörschel. ROX: Relational Over XML. Proc. of VLDB 2004, Toronto (2004)
10. M. Haustein, T. Härder. An Efficient Infrastructure for Native Transactional XML Processing. Data & Knowledge Engineering 61:3, 500-523, Elsevier, (2007)
11. T. Härder. The Layer Model and its Evolution. Datenbank-Spektrum, Heft 13, 45-57, 2005
12. T. Härder, M. Haustein, C.Mathis, M.Wagner. Node Labeling Schemes for Dynamic XML Documents Reconsidered. Data & Knowledge Engineering, 2006
13. A. Le Hors, P. Le Hégaret, L. Wood, G. Nicol, J. Robie, M. Champion, Steve Byrne. Document Object Model (DOM) Level 3 Core Specification, W3C Recommendation 2004
14. International Organization for Standardization (ISO). Information Technology–Database Language SQL-Part 14: XML-Related Specification (SQL/XML).
15. H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Yuquing Wu, Cong Yu. Timber: A native XML database. VLDB J., Volume 11, Number 4, p.274-291 (2002)
16. H.V. Jagadish, L.V.S Lakshmanan, D.Srivastava, K. Thompson. TAX: A Tree algebra for XML. In: Proc. of DBPL Conference, Rome, Italy. 2001
17. S. Klein. Professional SQL Server™ 2005 XML, Wiley Publishing, Inc. 2006
18. M. Krishnaprasad, Z. H. Liu, A. Manikutty, J. W. Warner, V. Arora, S. Kotsovolos. Query Rewrite for XML in Oracle XML DB. Proc. of the VLDB 2004.
19. Y. K. Lee, S. Yoo, K. Yoon, P. B. Berra. Index Structures for Structured Documents. In Proc. of ACM DL Conference, March 20-23 1996

20. Q Li, B. Moon. Indexing and Querying XML Data for Regular Path Expressions. Proc. of VLDB 2001, 2001
21. A. Maier and D. E. Simmen. DB2 Optimization in Support of Full Text Search, IEEE Data Engineering, Bull. 24(4), 2001
22. I. Manolescu, D. Florescu and D. Kossmann. Answering XML Queries on Heterogeneous Data Sources. VLDB 2001
23. N. M. Mattos. Integrating Information for On Demand Computing. Proc. of the 29th Int. Conf. on VLDB. 2003
24. W. Meier: eXist: An Open Source Native XML Database. LNCS 2593, 2003
25. C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, P. Schwarz. ARIES: a transaction recovery methods supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM Trans Database System 17(1):94-162. 1992
26. M. Nicola, B. van der Linden. Native XML Support in DB2 Universal Database. Proc. of the VLDB 2005
27. H. Pirahesh, J. M. Hellerstein, W. Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst, ACM SIGMOD 1992, pages 39-48
28. M. Rys. XML and Relational Database Management Systems: Inside Microsoft SQL Server™ 2005. Proc. of ACM SIGMOD 2005. 2005.
29. SAX Home Page. http://saxproject.org/. 2007
30. Dr. Harald Schöning. Tamino - A DBMS Designed for XML, Proceedings of 17th International Conference on Data Engineering, 2001
31. Shore – A High-Performance, Scalable, Persistent Object Repository. http://www.cs.wisc.edu/Shore
32. Database Language SQL – Part 2: Foundations (SQL/Foundations), ISO Final Draft International Standard, ISO 1998.
33. Xerces: a validating XML Parser. http://xml.apache.org/xerces-c/index.html
34. Xalan: an XSL Processor. http://xml.apache.org/xalan-c/index.html
35. XML 1.0. W3C Recommendation. 2006 http://www.w3.org/TR/REC-xml
36. XML Database Developer's Guide - Oracle XML DB. http://www.oracle.com