XML Databases and Beyond—
Plenty of Architectural Challenges Ahead

Theo Härder

University of Kaiserslautern, D-67653 Kaiserslautern, Germany.
haerder@informatik.uni-kl.de

Abstract. A key observation is that the invariants in database management determine the mapping steps of the supporting architecture. Referring to the multi-layered architecture of record-oriented database management systems (DBMSs), we sketch the advances made during the past decades. Then, we explore the ways how this proven architecture can be used to implement XML DBMSs (XDBMSs). Major changes and adaptations are needed in most of the layers to support fine-grained XML document processing (XDP). The use of DeweyIDs opens a new paradigm for the management of XML document trees: While preventing node relabeling, even in case arbitrary large subtrees are inserted into an XML document, DeweyIDs offer great benefits for efficient navigation in the document trees, for declarative query processing, and for fine-grained locking thereby avoiding access to external storage as far as possible. The proposed architecture also captures horizontal and vertical distribution of XML processing. Nevertheless, new architectural models are needed beyond record-oriented data types.

1 Introduction

Data independence is accomplished by the data model through set orientation and value-based, declarative requests together with the database management system (DBMS) implementing it. A high degree of logical and physical data independence is urgently needed to provide a flexible view mechanism thereby insulating the users, e.g., application programs, from DB schema evolution and to let the DBMS "survive" the permanent change in computer science in general and in the DB area in particular. Furthermore, DBMSs have a lifetime >20 or even >30 years. Therefore, far-reaching requirements concerning the extensibility and evolution of a DBMS are abundant: growing information demand led to enhanced standards with new object types, constraints, etc.; advances in research and development bred new storage structures and access paths, etc.; rapid changes of the technologies used and especially Moore's law strongly affected storage devices, memory, connectivity (e.g., Web), and so on.

We could already experience that a multi-layered hierarchical DBMS architecture is appropriate to fulfil the design objectives of data independence and to enable long-term system evolution and flexible extensibility as far as relational and object-relational data models and their implementations are concerned [8]. For this reason, we believe that it is a good starting point for architectural considerations of DBMSs beyond them.

Table 1 Description of the DBMS mapping hierarchy

	Level of abstraction	Objects	Auxiliary mapping data
L5	Nonprocedural or algebraic access	Tables, views, tuples	Logical schema description
L4	Record-oriented, navigational access	Records, sets, hierarchies, networks	Logical and physical schema description
L3	Record and access path management	Physical records, access paths	Free space tables, DB-key translation tables
L2	Propagation control	Segments, pages	DB buffer, page tables
L1	File management	Files, blocks	Directories, VTOCs, etc.

1.1 The History of the Layer Model

Mike Senko developed initial architectural concepts named Data Independent Accessing Model [18]. DIAM consists of four hierarchically layered levels called entity set model, string model, encoding model, and physical device level model. Some years later, Härder and Reuter refined these ideas and proposed a mapping model or reference architecture consisting of five hierarchical layers which should cooperate as "abstract machines" and achieve a high degree of information hiding among them to facilitate evolution and extensibility. As depicted in Table 1 [11], the architectural description embodies the major steps of dynamic abstraction from the level of physical storage up to the user interface. At the bottom, the database consists of huge volumes of bits stored on non-volatile storage devices, which are interpreted by the DBMS into meaningful information on which the user can operate. With each level of abstraction (proceeding upwards), the objects become more complex, allowing more powerful operations and being constrained by a growing number of integrity rules. The uppermost interface supports a specific data model, in our case by data access via SQL.

1.2 Major Extensions and Optimizations

While the explanation model concerning the DBMS architecture is still valid, an enormous evolution/progress has been made during the last two decades concerning functionality, performance, and scalability. The fact that all these enhancements and changes could be adopted by the proposed architecture, is a strong indication that we refer to a salient DBMS model. We cannot elaborate on all extensions, let alone to discuss them in detail, but we want to sketch some major improvements/changes.

Layer L1 was enhanced by the necessary functionality to attach and operate many new types of storage devices such as SSDs, Worms, DVDs. Furthermore, specialized

mapping functions allowed tailored clustering measures or LOB representation on external storage. Disk arrays with various forms of interleaving supported schemes with adjustable degrees of redundancy (e.g., the RAID project) and enabled declustering of objects at various levels to provide for parallel access.

At layer L2, Moore's Law increased the available memory for DB buffers by a factor of 10^4 within the past 20 years thereby achieving an optimization by default. Use of improved replacement algorithms—exploiting reference density combined with LRU (e.g., LRU-K)—, prefetching and pipelined execution in case of scan-based DB processing, etc. greatly improved DB buffer efficiency. Furthermore, configuring a set of buffers (for example, up to 80 in DB2) to separate workloads of different types and optimized to specific data types further boosted performance behavior at level L2.

Of all access path structures which could potentially fill level L3, the dominant one is still the ubiquitous B-tree. Despite a "firestorm" of research resulting in a few hundred proposals of novel index structures, the B- or B*-tree seem to be sufficient to cover all practical needs. At best, a few other structures such as UB-tree, R-tree, or Grid file are integration candidates for specialized access support. Indeed, the most dramatic performance enhancements at this architectural layer are due to fine-grained locking methods, in particular, applied to index structures, i.e., to B*-trees [16].

To mention a few optimization measures applied at level L4 and referring to the access paths of L3: selection and join algorithms utilizing TIDs of existing indexes thereby avoiding physical I/O as much as possible, hash joins which may also dramatically reduce access to external storage, and adaptive algorithms of various kinds which support load balancing and optimized throughput. Such adaptive techniques include setting or adjusting the degree of parallelism depending on the current workload, reordering and merging ranges to optimize repeated probes into an index, sharing scans among multiple queries, and so on [4].

Compilation and optimization of queries embodies the major functionality of L5. Although the quality of the optimizer—as a kind of landmark concept of a DBMS—has greatly improved in the course of the past two decades, e.g., by using refined statistics and histograms, there still remain open problems and even emerge new challenges. For example, user-defined types have to carry their own cost model to be integrated by cost-based optimizers. Furthermore, effective optimizers for dynamic QEPs (query execution plans) must address the problems of changes in resource availability or at least provide for dynamic plans with alternative algorithms or alternative plan shapes [4].

1.3 The Search for Future DBMS Architectures

The architectural layers sketched so far perfectly match the *invariants of set-oriented, record-like database management*: storage management (L1 and L2), access path and record management (L3), compilation, optimization, and evaluation of queries (L4 and L5). During the recent decade, integration efforts for functionality not fitting into this framework were primarily based on a kind of loose coupling of components—called Extenders, DataBlades, or Cardridges—and a so-called extensibility infrastructure. Because these approaches could neither fulfil the demands for seamless integration nor the

overblown performance and scalability expectations, future solutions may face major changes in the architecture.

A hot topic of research is the appropriate integration of XML document management, because messages are data, too. Questions controversially discussed so far are "Will the DBMSs of the future be hybrids, storing both relational and XML?" or "Will everything be stored in XML format?" making myriads of SQL systems "legacy applications". Besides hybrid architectures which map XML documents and tables by separate storage and access systems and support coexistence/combination of DB requests of both kinds, a futuristic scenario motivated by the questions above was discussed in *ROX: Relational over XML* [15] to support SQL APIs as well as XDP interfaces. While XML operations on native XML structures are the target of optimization in XDBMSs, such future DBMS architectures represent mixed SQL and XQuery systems to run SQL applications on native XML or on hybrid structures concurrently.

A key observation of relational DBMS architectures is that the *invariants in database management determine the mapping steps of the supporting architecture*. Because of the record-oriented nature of fine-grained management of XML documents the invariants of XDP are, at least, similar to the relational ones. Therefore, we explore in Sect. 2 the ways how the original layer model has to be adjusted to serve for the description and explanation of XDBMS implementations. In Sect. 3, we consider variants of this model to be applied to data management scenarios where horizontal and vertical distribution of XML database processing is needed. Sect. 4 sketches a number of new data types which cannot be smoothly integrated into the architectural framework and argues about the need for enhanced adaptivity and dependability properties for future DBMSs. Finally, we conclude with some brief remarks in Sect. 5.

2 Architectural Requirements for XML Databases

Currently available relational or object-relational (O)RDBMSs only manage structured data well. There is no effective and straightforward way for handling XML data. This is obviously true when simple CLOB types have to be used. In particular, searching of XML documents becomes prohibitively slow. But also more refined mappings do not lead to good solutions per se: An innumerable number of algorithms [19] has been proposed for the mapping of semi-structured XML data to structured relational database tables and columns (the so-called "shredding"). All these approaches have failed to efficiently support the wide spectrum of DB applications and to guarantee satisfactory performance in high-performance transaction environments. Furthermore, as XML documents permeate information systems and databases with increasing pace, they are more and more used in a collaborative way. If you run today an experiment on existing DBMSs with collaborative XML documents, you may experience a "performance catastrophe" meaning that most transactional operations are processed in strict serial order. The challenge for database system development is to provide adequate and fine-grained management for these documents enabling efficient and concurrent read and write operations. Therefore, future XML DBMSs will be judged according to their ability to achieve high transaction parallelism.

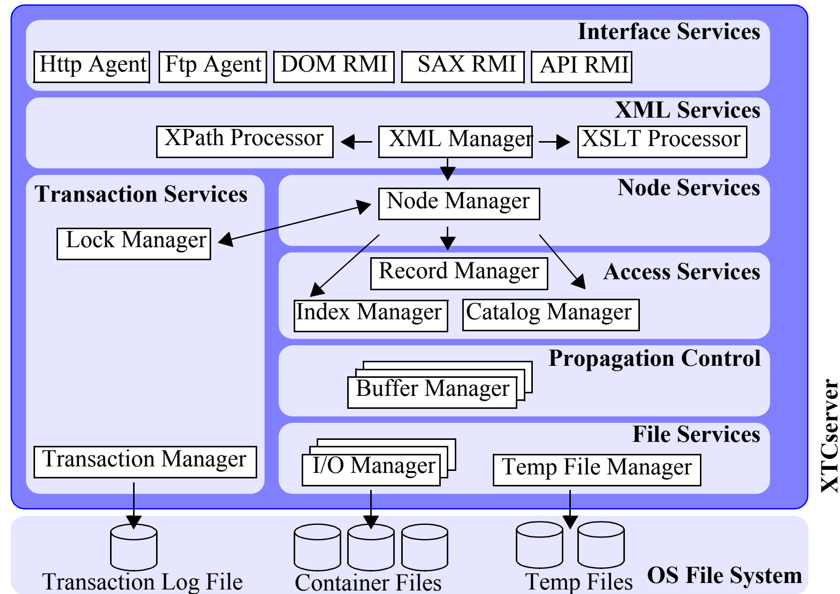

Fig. 1 XTC system – overview

2.1 XTC Architecture

First attempts to provide for DB-based XML processing focused on using the lower layer features of relational DBMSs such that roughly the access and storage system layers were *reused and complemented* by the data system functionality tailored to the demands of the XML data model (e.g., DOM, SAX, XQuery); this implied the mapping (called "shredding") of XML document structures onto a set of tables.

Although viable within our five-layer architecture (by reusing L1 to L4), this idea had serious performance trade-offs, mainly in the areas of query optimization and concurrency control. New concepts and implementation techniques in the reused layers are required to achieve efficient query processing. For these reasons, so-called native XML DBMSs emerged in recent years, an architectural example of which is illustrated in Fig. 1. The current state of the XTC architecture (XML Transaction Coordinator [12]) perfectly proves that native XDBMSs can be implemented along the lines of our five-layer architecture.

2.2 Storage and Buffer Management

At the layers L1 and L2, reuse of concepts as described in Sect. 1.2 is obvious. Hence, we can more or less adopt the mechanisms proven in relational DBMS implementations and adjust them to the specific needs of XML document representations. In summary,

our storage layer offers an extensible file structure based on the B*-tree mechanism as a container of single XML documents such that updates of an XML document (by IUD operations) can be performed on any of its nodes. We have shown that a very high degree of storage occupancy (> 96%) for XML documents is achieved under a variety of different update workloads.

Although the functionality in the remaining three layers is comparable at an abstract level, the objects and the specific implementation methods exhibit strong distinctions. Due to space restrictions, we can only focus on some new important aspects.

2.3 Access Services

Efficient and effective processing and concurrent operations on XML documents are greatly facilitated, if we use a specialized internal representation which enables fine-granular management and locking. For this reason, we have implemented in our XTC system the taDOM storage model illustrated in Fig. 3 as a slight extension of the XML tree representation defined in [21]. In contrast to the DOM tree, we do not directly attach attributes to their element node, but introduce separate *attribute roots* which connect the attribute nodes to the respective elements. String nodes are used to store the actual content of an attribute or a text node. Via the DOM API, this separation enables access of nodes independently of their value. Our representational enhancement does not influence the user operations and their semantics on the XML document, but is solely exploited by the lock manager to achieve certain kinds of optimizations.

Most influential for an access model to the tree nodes of an XML document is a suitable node labeling scheme for which several candidates have been proposed in the literature. While most of them are adequate to label static XML documents, the design of schemes for dynamic documents allowing arbitrary insertions within the tree—free of reorganization, i.e., no reassignment of labels to existing nodes—remains a challenging research issue. The existing approaches can be classified into range-based and prefix-based labeling schemes. While range-based schemes consisting of independent numbering elements (e.g., DocID, startPos : endPos, level, see [1]) seem to be less amenable to algorithmic use and cannot always avoid relabeling in case of node insertions, prefix-based schemes seem to be more flexible. We believe that they are at least as expressive as range-based schemes, while they guarantee stability of node IDs under arbitrary insertions, in addition. In particular, we favor a scheme supporting efficient insertion and compression while providing the so-called Dewey order (defined by the Dewey Decimal Classification System). Conceptually similar to the ORDPATH scheme [17], our scheme refines the mapping and solves practical problems of the implementation.

Fast access to and identification of all nodes of an XML document is mandatory to enable effective indexing primarily supporting declarative queries and efficient processing of direct-access methods (e. g., *getElementById()*) as well as navigational methods (e. g., *getNextSibling()*). For this reason, we have implemented the node labeling scheme whose advantages should be illuminated by referring to Fig. 2. For example, a DeweyID is 1.3.4.3.5 which consists of several so-called *divisions* separated by dots (in the human readable format). The root node of the document is always labeled by Dew-

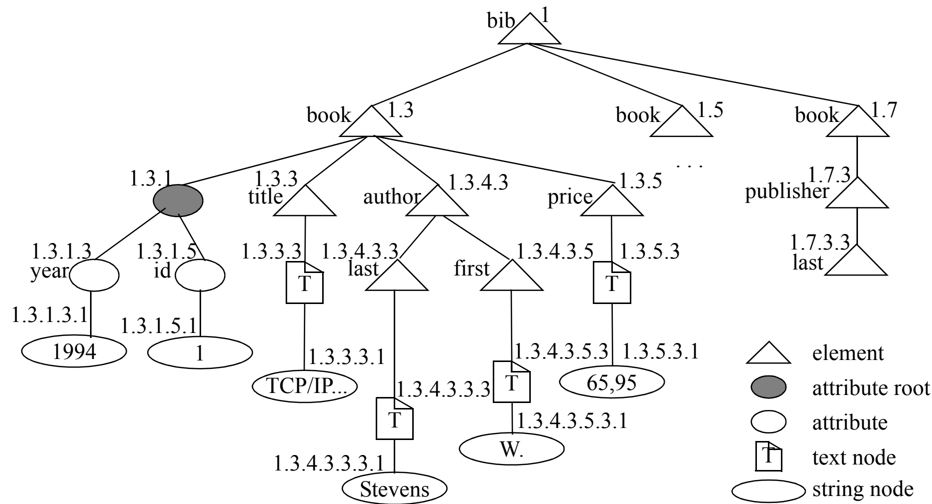

Fig. 2 A sample taDOM tree labeled with DeweyIDs

eyID 1 and consists of only a single division. The children obtain the DeweyID of their parent and attach another division whose value increases from left to right. To allow for later node insertions at a given level, we introduce a parameter *distance* which determines the gap initially left free in the labeling space. In Fig. 3, we have chosen the minimum distance value of 2. Furthermore, assigning at a given level a distance to the first child, we always start with *distance + 1*, thereby reserving division value 1 for attribute roots and string nodes (illustrated for the attribute root of 1.3 with DeweyID 1.3.1). Hence, the mechanism of the Dewey order is quite simple when the IDs are initially assigned, e.g., when all nodes of the document are bulk-loaded. In the above tree example, the author node is inserted later within the gap 1.3.3 to 1.3.5. Because arbitrary many nodes may be inserted into any gap, we need a kind of overflow mechanism indicating that the labeling scheme remains at the same level when an odd division value is not available anymore for a gap. Thus, we reserve even division values for that purpose; they may occur consecutively (depending on the insertion history) where an uninterrupted sequence of even values just states that the same labeling level is kept [13].

The salient features of a scheme assigning a DeweyID to each tree node include the following properties: Referring to the DeweyID of a node, we can determine the level of the node in the tree and the DeweyID of the parent node. Hence, we can derive its entire ancestor path up to the document root without accessing the document. By comparing the DeweyIDs of two nodes, we can decide which node appears first in the document's node order. If all sibling nodes are known, we can determine the exact position of the node within the document tree. Furthermore, it is possible to insert new nodes at arbitrary locations without relabeling existing nodes. In addition, we can rapidly figure out all nodes accessible via the typical XML navigation steps, if the nodes are stored in document order, i.e., in left-most depth-first order.

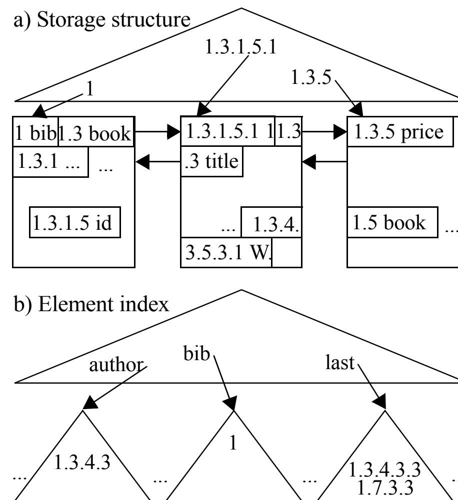

Fig. 3 Document storage using B*-trees

Fast (indexed) access to each node is provided by variants of B*-trees tailored to our requirements of node identification and direct or relative location of any node. Fig. 3a illustrates the storage structure—consisting of *document index* and *document container* as a set of chained pages—for the sample XML document of Fig. 2, which is stored in document order; the key-value pairs within the document index are referencing the first DeweyID stored in each container page. Additionally to the storage structure of the actual document, an *element index* is created consisting of a *name directory* with all element names occurring in the XML document (Fig. 3b); for each specific element name, in turn, a *node-reference index* is maintained which addresses the corresponding elements using their DeweyIDs. In all cases, variable-length key support is mandatory; additional functionality for prefix compression of DeweyIDs is very effective. Because of reference locality in the B*-trees while processing XML documents, most of the referenced tree pages (at least the ones belonging to the upper tree layers) are expected to reside in DB buffers—thus reducing external accesses to a minimum.

2.4 Node Services—Support of Navigation, Query Evaluation, and Locking

Selection and join algorithms based on index access via TID lists together with the availability of fine-grained index locking boosted the performance of DBMSs [8], because they reduced storage access and minimized blocking situations for concurrent transactions as far as possible. Both factors are even more critical in XDBMS. Hence, when designing such a system, we have to consider them very carefully.

Using the document index sketched in Fig. 3, the five basic navigational axes *parent, previous-sibling, following-sibling, first-child,* and *last-child*, as specified in DOM [21], may be efficiently evaluated—in the best case, they reside in the page of the given context node *cn.* When accessing the previous sibling *ps* of *cn*, e.g., node 1.5 in Fig. 2, an obvious strategy would be to locate the page of 1.5 requiring a traversal of the document index from the root page to the leaf page where 1.5 is stored. This page is often already present in main memory because of reference locality. From the context node, we check all IDs backwards, following the links between the leaf pages of the index, until we find *ps*—the first ID with the same parent as *cn* and the same level. All IDs skipped along this way were descendants of *ps*. Therefore, the number of pages to be accessed depends on the size of the subtree having *ps* as root. An alternative strategy avoids this unwanted dependency: After the page containing 1.5 is loaded, we inspect

the ID *d* of the directly preceding node of 1.5, which is 1.3.5.3.1. If *ps* exists, *d* must be a descendant of *ps*. With the level information of *cn*, we can infer the ID of *ps*: 1.3. Now a direct access to 1.3 suffices to locate the result. The second strategy ensures independence from the document structure, i.e., the number of descendants between *ps* and *cn* does not matter anymore. Similar search algorithms for the remaining four axes can be found. The *parent* axis, as well as *first-child* and *next-sibling* can be retrieved directly, requiring only a single document index traversal. The *last-child* axis works similar to the *previous-sibling* axis and, therefore, needs two index traversals in the worst case.

For declarative access via query languages like XQuery, a set-at-a-time processing approach—or more accurately, sequence-at-a-time—and the use of the element index promise in some cases increased performance over a navigational evaluation strategy. Nevertheless, the basic DOM primitives are a fallback solution, if no index support is available. To illuminate the element index use for declarative access, let us consider a simple XQuery predicate that only contains forward and reverse step expressions with name tests: *axis1::name1/.../axisN::nameN*. XQuery contains 13 axes, 8 of which span the four main dimensions in an XML document: *parent–child, ancestor–descendant, preceding-sibling–following-sibling,* and *preceding–following*. For each axis, we provide an algorithm that operates on a duplicate-free input sequence of nodes in document order and produces an output sequence with the same properties and containing for the specified axis all nodes which passed the name test. Therefore, the evaluation of axes is closed in this group of algorithms and we can freely concatenate them to evaluate path expressions having the referenced structure. Our evaluation strategy follows the idea of structural joins [1] adjusted to DeweyIDs, and additionally expanded to support the *preceding-sibling–following-sibling* and *preceding–following* dimensions.

Let us consider the *following-sibling* axis as an example. In Fig. 4, the nodes of the input sequence P, which may be the result of a former path step, are marked in a dark shade. Furthermore, the sequence of nodes F in our document that satisfy the name test for the current evaluation of the *following-sibling* axis carry the letter 'n'. The DeweyIDs of these nodes are retrieved using the element index. A problem of using the *following-sibling* axis is the possible generation of duplicates. For example, node 1.3.9 qualifies as a *following-sibling* for nodes 1.3.3, 1.3.5, and 1.3.7. Because duplicate removal is an expensive operation, our strategy is to avoid duplicates in the first place. The evaluation algorithm works as follows: In a first phase, input P is processed in document order. For each DeweyID *d*, a pair (key, value) as *(parent(d), d)* is added to a hash table HT. If

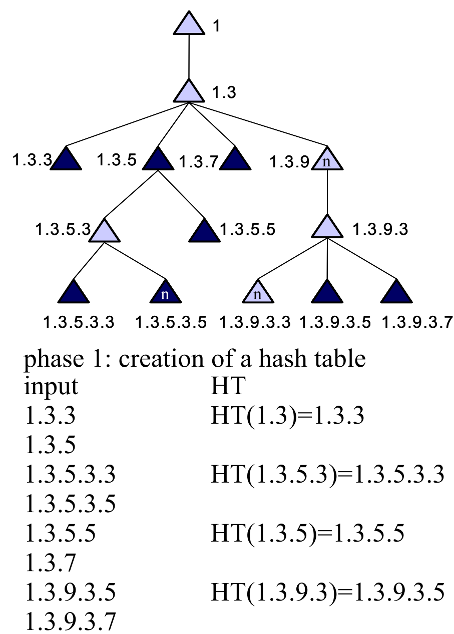

Fig. 4 Following-sibling algorithm

parent(d) is already present in HT, *d* can be skipped. Because we process P in document order, only the first sibling among a group of siblings is added to HT. In the second phase, we iterate over F and probe each ID *f* against HT. If *parent(f)* is contained in HT, we simply compare whether or not *f* is a *following-sibling* of HT*(parent(f))*. This comparison can easily be done by looking at the two DeweyIDs. Assume, the parent of *f*=1.3.5.3.5 is contained in HT and *f* is a *following-sibling* of HT(1.3.5.3), then *f* will be included into the result sequence. For ID *f*=1.3.9.3.3, this test fails, because *f* is not a *following-sibling* of 1.3.9.3.5. F is processed in document order, therefore, the output also obtains this order. Similar evaluation algorithms are provided for all other axes.

Fine-grained concurrency control is of outmost importance for collaborative use of XML documents. Although predicate locking of XQuery and XUpdate-like statements [21] would be powerful and elegant, its implementation rapidly leads to severe drawbacks such as the need to acquire large lock granules, e.g., for predicate evaluations as shown in Fig. 4, and undecidability problems—a lesson learned from the (much simpler) relational world. To provide for a multi-lingual solution, we necessarily have to map XQuery operations to a navigational access model to accomplish fine-granular concurrency control. Such an approach implicitly supports other interfaces such as DOM, because their operations correspond more or less directly to a navigational access model. Therefore, we have designed and optimized a group of lock protocols explicitly tailored to the DOM interfaces which are absolutely complex—20 lock modes for nodes and three modes for edges together with the related compatibilities and conversion rules—, but for which we proved their correctness [12] and optimality [1][14].

2.5 Query Compilation and Optimization

The prime task of layer L5 is to produce QEPs, i.e., to translate, optimize, and bind the multi-lingual requests—declarative as well as navigational—from the language models to the operations available at the logical access model interface (L4). For DOM and SAX requests, this task is straightforward. In contrast, XQuery or XPath requests will be a great challenge for cost-based optimizers for decades. Remember, for complex languages such as SQL:2003 (simpler than the current standard of XQuery), we have experienced a never-ending research and development history—for 30 years to date—and the present optimizers still are far from perfect. For example, selectivity estimation is much more complex, because the cardinality numbers for nodes in variable-depth subtrees have to be determined or estimated. Furthermore, all current or future problems to be solved for relational DBMSs [4] will occur in XDBMSs, too.

---

1. By using so-called meta-synchronization, XTC maps the meta-lock requests to the actual locking algorithm which is achieved by the lock manager's interface. Hence, exchanging the lock manager's interface implementation exchanges the system's complete XML locking mechanism. In this way, we could run XTC in our experiments with 11 different lock protocols. At the same time, all experiments were performed on the taDOM storage model optimized for fine-grained management of XML documents.

3 Architectural Variants

Because the invariants in database management determine the mapping steps of the supporting architecture, we can also use our architectural framework in new data management scenarios where XDBMSs are involved, as long as the basic invariants still hold true: page-oriented mapping to external storage, management of record-oriented data, set-oriented/navigational data processing. Similar to the scenarios evolved in the past for relational database management, equivalent ones may emerge in the future, in case XDBMSs gain the momentum in the market.

3.1 Horizontal Distribution of XDBMS Processing

A variety of DB processing scenarios can be characterized as the horizontal distribution of the entire DB functionality and of partitioned/replicated data to processing nodes connected by a network. As a consequence, the core requirements remain, leading to a simplified architectural model sketched in Fig. 5, which consists of identical layered models for every node together with a *connection layer* responsible for communication, adaptation, or mediation services. In an implementation, this layer could be integrated with one of the existing layers or attached to the node architecture to encapsulate it for the remaining system.

For these reasons, our layer model can serve as a framework for the implementation of XDBMS variants for architectural classes such as *Shared Nothing,* S*hared Disk, and Parallel DBMSs,* because all of them have to run identical operations in the various layers. Adaptation of processing primarily concerns the handling of partitioning or replication and, as a consequence, issues of invalidation, synchronization, and logging/recovery.

When heterogeneity of the data models or autonomy of database systems comes into play, the primary tasks of the connection layer are concerned with adaptation and mediation. *Federated XDBMSs* could represent the entire spectrum of possible data integration scenarios and would need an adjustment of the DB requests at the level of the data model or a compensation of functionality not generally available. As opposed to distributed homogeneous XDBMSs, some users (transactions) may only refer to a local view thereby abstaining from federated services, while, at the same time, other users exploit the full services of the data federation. The other extreme case among the federation scenarios is represented by *Multi-XDBMSs*, for which the connection layer primarily takes over the role of a global transaction manager passing unmodified DB requests to the participating DB servers.

3.2 Vertical Distribution of XDBMS Processing

Our layer model also fits to client/server database processing. In this category, the major concern is to make XDBMS processing capacity available close (or at least closer) to the application of the client (computer). So far, client/server DBMSs are used in appli-

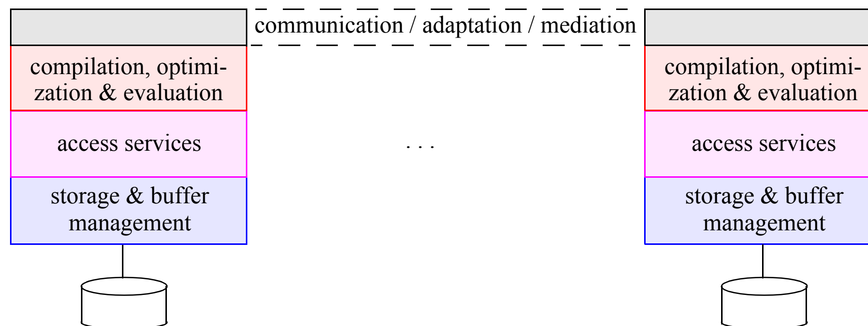

Fig. 5 Horizontal XDBMS distribution

cations relying on long-running transactions with a checkout/checkin mechanism for (versioned) data. Hence, the underlying data management scenarios are primarily tailored to engineering applications. Object-oriented DBMS distinguish between file servers, object servers, and query servers: the most sophisticated ones are the query servers. Their real challenge is declarative, set-oriented query processing thereby using the current content of the query result buffer [3].

Until recently, query processing in such buffers was typically limited to queries with predicates on single tables (or equivalent object types). Now, a major enhancement is pursued in scenarios called *database caching*. Here, full-fledged DBMSs, used as frontend DBs close to application servers in the Web, take over the role of cache managers for a backend DB. As a special kind of vertical distribution, their performance-enhancing objective is to evaluate more complex queries in the cache which, e.g., span several tables organized as cache groups by equi-joins [10]. The magic concept is *predicate completeness* where the DBMS (i.e., its cache manager) has to guarantee that all objects required for the evaluation of a query predicate are present in the cache and are consistent with the DB state in the backend DB. So far, these concepts are explored for relational models, e.g., SQL. However, we have observed that the idea of predicate completeness can be extended to other types of data models—in particular, XML data models—, too. Thinking about the potential of this idea gives us the vision that we could support the entire user-to-data path in the Internet with a single XML data model [9].

While the locality preservation of the query result buffer in query server architectures can take advantage of application hints [3], *adaptivity* of database caching is a major challenge for future research [2]. Furthermore, precise specification of *relaxed currency and consistency* of data is an important future task to better control the widespread and growing use of distant caches and asynchronous copies [7]. Other interesting research problems occur if transactional updates are directly performed in DB caches. Instead of processing them in the backend DB first, they could be executed in the cache or even jointly in cache and backend DB under a 2PC protocol. Such update models may lead to futuristic considerations where the conventional hierarchic arrangement of frontend cache and backend DB is dissolved: If each of them can play both roles and if together they can provide consistency for DB data, more effective DB support may be gained for new applications such as grid or P2P computing.

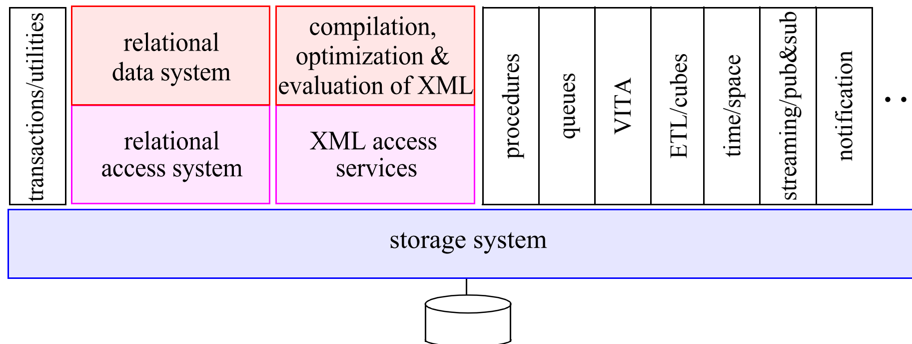

Fig. 6 Desirable extensions for future DBMS architectures

4 New Types of DBMS Architectures?

XML data could not be adequately integrated into the original layer model because the processing invariants valid in record-oriented DBMS do not hold true for document trees with other types of DB requests. Therefore, we needed substantial changes and adaptations, especially in layers L3 to L5, while the overall layered framework could be preserved. However, what has to be done when the conceptual differences of the data types such as VITA (video, image, text, audio) or data streams are even larger?

4.1 The Next Database Revolution Ahead?

VITA types, for example, are managed in tailored DB buffers and are typically delivered (in variable-length junks) to the application thereby avoiding additional layer crossings. In turn, to avoid data transfers, the application may pass down some operations to the buffer to directly manipulate the buffered object representation. Hence, Fig. 6 illustrates that the OS services or, at best, the storage system represent the least common denominator for the desired DBMS extensions.

If the commonalities in data management invariants for the different types and thus the reuse opportunities for functionality are so marginal, it makes no sense to squeeze all of them into a unified DBMS architecture. As a proposal for future research and development, Jim Gray sketched a framework leading to a diversity of type-specific DBMS architectures [6]. As a consequence, we obtain a collection of heterogeneous DBMSs which have to be made accessible for the applications—as transparently as possible by suitable APIs. Such a collection embodies an “extensible object-relational system where non-procedural relational operators manipulate object sets. Coupled with this, each DBMS is now a Web service” [6]. Furthermore, because they cooperate on behalf of applications, ACID protection has to be assured for all messages and data taking part in a transaction.

4.2 Dependability versus Adaptivity

Orthogonal to the desire to provide functional extensions, the key role of DBMSs in modern societies places other kinds of "stress" on their architecture. Adaptivity to application environments with their frequently changing demands in combination with dependability in critical situations will become more important design goals—both leading to contradicting guidelines for the architectural design.

So far, information hiding and layers as abstract machines were the cornerstones for the design of large evolutionary DBMSs. Typically, adaptable component (layer) behavior cannot be achieved by exploiting local "self"-observations alone. Hence, autonomic computing principles applied to DBMS components require more information exchange across components (introducing more dependencies) to gain a more accurate view when decisions relevant for behavioral adaptations have to be made. Trouble-free operation of a DBMS primarily comes from adjustment mechanisms automatically applied to problems of administration, tuning, coordination, growth, hardware and software upgrades, etc. Ideally, the human system manager should only set goals, policies, and a budget while the automatic adaptation mechanisms should do the rest [5]. Online feedback control loops are key to achieve such adaptation and "self-*" system properties, which, however, amplify the information channels across system layers.

In contrast, too many information channels increase the inter-component complexity and are directed against salient software engineering principles for highly evolutionary systems. In this respect, they work against the very important dependability objective which is much broader than self-tuning or self-administration. Hence, design challenges are to develop a system which should be always available, i.e., exhibiting an extremely high availability, and which only services authorized uses, i.e., even hackers cannot destroy data or force the system to deny services to authorized users. Jim Gray summarizes the main properties of a dependable and adaptive system as *always-up* + *secure* + *trouble-free*. To develop such systems, innovative architectures observing new software engineering principles have to be adopted. However, most of their properties are not easily amenable to mathematical modeling and runtime analysis, because they are non-functional in general. Weikum calls for a highly componentized system architecture with small, well-controlled component interfaces and limited and relatively simple functionality per component which implies the reduction of optional choices [20]. The giant chasm to be closed results from diverging requirements: *growing system complexity* due to new extensions and improved adaptivity as opposed to *urgent simplification needs* mandatory for the development of dependable systems.

5 Conclusions

In this paper, we primarily explored how XDBMSs fit into the framework of a multi-layered hierarchical architecture originally developed for record-oriented data models. We proposed major changes and adaptations for which DeweyIDs embody the fundamentally new concept. Their expressive power and stability enabled new classes of evaluation algorithms for services supporting navigation, declarative queries, and fine-

grained locking. Finally, we sketched some ideas for integration data types which cannot be efficiently mapped to the layer architecture and emphasized the need to decidedly improve adaptability and dependability properties in future DBMSs.

References

[1] S. Al-Khalifa, et al.: Structural Joins: A Primitive for Efficient XML Query Pattern Matching. Proc. 18th Int. Conf. on Data Engineering, 141 (2002)

[2] M. Altinel et al.: Cache Tables: Paving the Way for an Adaptive Database Cache. VLDB 2003: 718-729

[3] S. Deßloch, T. Härder, N. M. Mattos, B. Mitschang, J. Thomas: Advanced Data Processing in KRISYS. VLDB J. 7(2): 79-95 (1998)

[4] G. Graefe: Dynamic Query Evaluation Plans: Some Course Corrections? IEEE Data Eng. Bull. 23(2): 3-6 (2000)

[5] J. Gray.: What next?: A dozen information-technology research goals. J. ACM 50(1): 41-57 (2003) (Journal Version of the 1999 ACM Turing Award Lecture)

[6] J. Gray: The Next Database Revolution. SIGMOD Conference 2004: 1-4

[7] H. Guo, P.-A. Larson, R. Ramakrishnan, J. Goldstein: Relaxed Currency and Consistency: How to Say "Good Enough" in SQL. SIGMOD Conference 2004: 815-826

[8] T. Härder: DBMS Architecture—Still an Open Problem. Proc. Datenbanksysteme in Business, Technologie und Web (BTW 2005), LNI P-65, Springer, 2-28, 2005

[9] T. Härder: Caching over the Entire User-to-Data Path in the Internet, in: T. Härder, W. Lehner (eds), Data Management in a Connected World, LNCS 3551, 2005, pp. 67–89

[10] T. Härder, A. Bühmann: Query Processing in Constraint-Based Database Caches. Data Engineering Bulletin 27:2 (2004) 3-10

[11] T. Härder, A. Reuter: Concepts for Implementing a Centralized Database Management System. Proc. Int. Comp. Symp. on Appl. Systems Development, 1983, Nürnberg, 28-60

[12] M. Haustein, T. Härder: Optimizing Concurrent XML Processing, submitted (2005)

[13] M. Haustein, T. Härder, C. Mathis, M. Wagner: DeweyIDs—The Key to Fine-Grained Management of XML Documents, submitted (2005)

[14] M. Haustein, T. Härder, K. Luttenberger: Contest of Lock Protocols—The Winner is taDOM3+, submitted (2005), http://wwwdvs.informatik.uni-kl.de/pubs/p2005.html

[15] A. Halverson, V. Josifovski, G. Lohman, H. Pirahesh, M. Mörschel. ROX: Relational Over XML. Proc. 30th Int. Conf. on Very Large Data Bases, Toronto (Sept. 2004)

[16] C. Mohan: ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes. VLDB 1990: 392-405

[17] P. E. O'Neil et al.: ORDPATHs: Insert-Friendly XML Node Labels. Proc. SIGMOD Conf.: 903-908 (2004)

[18] M. E. Senko, E. B. Altman, M. M. Astrahan,P. L. Fehder: Data Structures and Accessing in Data Base Systems. IBM Systems Journal 12(1): 30-93 (1973)

[19] I. Tatarinov et al.: Storing and Querying Ordered XML Using a Relational Database System. Proc. ACM SIGMOD, Madison, Wisconsin, USA, 204-215 (2002)

[20] G. Weikum, A. Mönkeberg, C. Hasse, P. Zabback: Self-tuning Database Technology and Information Services: from Wishful Thinking to Viable Engineering. VLDB 2002: 20-31

[21] W3C Recommendations. http://www.w3c.org (2004)