Theo Härder

DBMS Architecture – New Challenges Ahead

More than two decades ago, an architectural model based on successive data abstraction steps of record-oriented data was proposed as kind of a standard and later refined to a five-layer hierarchical DBMS model. While this model greatly supported all requirements of horizontal and vertical distribution of record-oriented DB processing, it is now challenged by a variety of new requirements and changes as far as processing environments, data types, functional extensions, heterogeneity, autonomy, etc. are concerned. We discuss some cases where the original layer model can only keep up through substantial changes in the individual layers and where totally new architectural models have to be found. Furthermore, a broader perspective is needed when dependable adaptive information systems are designed. In this respect, we will identify a big chasm resulting from diverging requirements and leading to a conflict in the design objectives: growing system complexity due to extensions in current DBMSs which will be augmented by improving adaptivity as opposed to urgent simplification needs mandatory for the development of dependable systems.[1]

1 Motivation

In [Härder 2005c], we have intensively discussed the issues of data mapping and transaction support in a centralized DBMS architecture consisting of five hierarchical layers. Furthermore, we have outlined how the layered architecture could be adjusted to progress in science and technology evolving in the last two decades, that is, our architectural model successfully responded to the pressure of »permanent change«. It could even prove its extensibility potential for new requirements of a variety of emerging data management tasks. By introducing an additional layer for the needs of communication, adaptation, and mediation, our architectural model can also serve for scenarios where DB processing has to be distributed in a horizontal or vertical fashion.

Distributed DBMSs are the main exponent of horizontal distribution of the entire DB functionality and of partitioned/replicated data to processing nodes connected by a network [Rahm 1994]. As a consequence, the core requirements remain, leading to an architectural model which consists of identical layered models for every node together with a *connection layer* responsible for communication, adaptation, or mediation services. In contrast, vertical distribution is typically achieved by so-called *client/server DBMSs*. Their major concern is to make DBMS processing capacity available close to the application in the client (computer). Usually, client/server DBMSs are used in applications relying on long-running transactions with a checkout/checkin mechanism for (versioned) data.

In all of these cases, however, navigational or set-oriented processing of record-like data was the primary objective to be supported by the layered architecture, as sketched in Table 1. We have observed that the *invariants in database management determine the mapping steps of the supporting architecture* and, hence, an architectural model serves well as long as it can effectively support these basic invariants. For the appropriate use of the layered architecture, its basic invariants should hold true: page-oriented mapping to external storage, management of record-oriented data, set-oriented database processing.

Today, however, each DBMS architecture is flooded by a wave of requirements for new data types, transactional concepts, and data management scenarios where these invariants only partially hold or where they have to be replaced by totally new concepts. Nowadays important DBMS requirements include data streams, unstructured or semi-structured documents, time series, spatial objects, etc. Due to the variety of these emerging and diverging demands and application trends, we should seriously explore the question whether the evolutionary potential of our architectural model is sufficient to adopt the new functionality or whether we need »a revolution in the area of DBMS architecture« [Gray 2004]?

In this paper, we explore new architectural requirements and try to answer this question. For this reason, we start in section 2 with moderate deviations from the traditional data management invariants and debate urging issues of native DB processing of XML documents. Database caching sketched in section 3 focusses on declarative and set-oriented query processing in caches close to the application servers sitting at the edge of the Web. By this vertical distribution, the backend DBMS can be unburdened to increase overall system scalability and performance. Furthermore, user-perceived latency for dynamically created Web objects may be greatly improved. In section 4, we consider the role of our architectural model when applied to other kinds of data management scenarios and sketch extensions to adjust for a variety of new data types. We illustrate the reasons why the »next DBMS revolution« may be around the corner, because a large variety of database services for an »ecosystem« of new data types is urgently needed.

Because information system services are ubiquitous in our daily life, adaptivity and dependability of data management services have to be substantially enhanced. For this reason, we have to consider a bigger picture in section 5 including the information system perspective as an ecosystem consisting of DBMS engine(s), DB-related middleware, and application integration, before we briefly summarize our contribution in section 6.

Table 1: DBMS mapping hierarchy

	Level of abstraction	**Objects**
L5	Nonprocedural access	Tables, views, tuples
L4	Navigational access	Logical records, sets, networks
L3	Access path mgmt	Physical records, access paths
L2	Propagation control	Segments, pages
L1	File mgmt	Files, blocks

1. This contribution is the second part of an extended version of [Härder 2005a and 2005b].

2 Native DB Processing of XML Documents

For the data management scenarios sketched above, the layers of our architectural model (see Table 1) perfectly match the invariants of set-oriented, record-like database management such that they could be reused more or less unchanged in the outlined DBMS variants. However, recent requirements strongly deviate from this processing paradigm. Integration efforts developed during the last 10 years were primarily based on a kind of loose coupling of components – called Extenders, DataBlades, or Cartridges – 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.

2.1 Architectural Layer Reuse

First attempts to provide for DB-based XML processing focused on using the lower layer features of relational DBMSs (RDBMSs) 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 proceeding implied the mapping (called »shredding«) of XML document structures onto a set of tables for which numerous proposals were published [Florescu & Kossmann 1999].

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[2]. For these reasons, so-called native XML DBMSs (XDBMSs, [Jagadish et al 2002]) emerged in recent years, an architectural example of which is illustrated in Fig. 1.

Here we use XTC (XML Transaction Coordinator, [Haustein 2005]) as an example. It is a full-fledged native XDBMS which initially was developed to serve as a testbed system for exploring and evaluating fine-grained concurrency control on

2. »A growing number of application developers believe XML and XQuery should be treated as our primary data structure and access pattern« [Gray 2005].

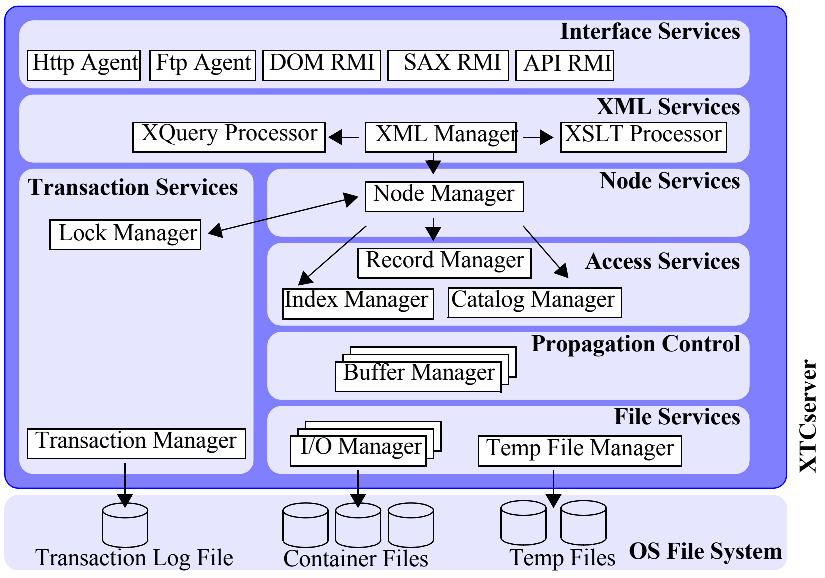

Fig. 1: XTC system – overview

XML document processing (XDP) via some/all of the standardized XML language interfaces [DOM 2004, XQuery 2004]. Its current state 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 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. While we use DOM trees – containing element, attribute, and text nodes as defined in [DOM 2004] –, for the representation of XML documents on external storage, we have implemented for their memory representation a slight extension, the so-called taDOM storage model [Haustein 2005]. 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 for 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

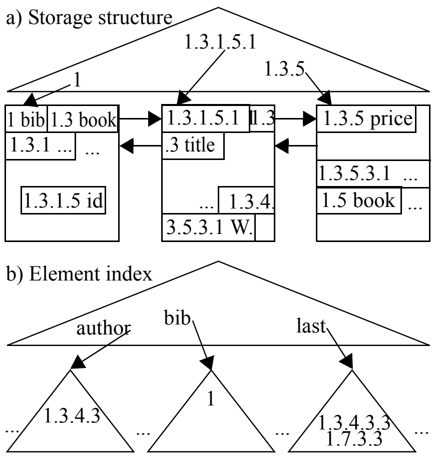

Fig. 2: Document storage using B*-trees

range-based schemes consisting of independent numbering elements (e.g., DocID, startPos : endPos, level, see [Al-Khalifa et al. 2002]) 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).

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()*). Conceptually similar to the ORDPATH scheme [O'Neil et al. 2004], our DeweyID scheme refines the Dewey order mapping and introduces a kind of overflow mechanism when »gaps« for new insertions are in short supply in the labeling space. A DeweyID consists of several so-called *divisions* separated by dots (in the human readable format). The root node of the document is always labeled by DeweyID 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. An empirical evaluation of this scheme can be found in [Haustein & al. 2005b].

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 leftmost depth-first order. Nevertheless, DeweyIDs tend to become quite long, depending on the depth of the document, the distance parameter, and possible »gap« overflows. Therefore, suitable encoding and compression schemes for their implementation are mandatory.

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. 2a illustrates the storage structure – consisting of *document index* and *document container* as a set of chained pages – sketching a sample XML document, which is stored in document order; the key-value pairs within the document index are referencing the first DeweyID stored in each container page. In addition to the storage structure of the actual document, an *element index* is created consisting of a *name directory* with (potentially) all element names occurring in the XML document (Fig. 2b); for each specific element name, in turn, a *node-reference index* may be 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 [Härder 2005a], 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. 2, the five basic navigational axes *parent, previous-sibling, following-sibling, first-child,* and *last-child*, as specified in DOM [DOM 2004], may be efficiently evaluated – in the best case, the corresponding objects reside in the page of the given context node *cn.* When accessing the previous sibling *ps* of *cn*, e.g., of 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. Hence, we inspect the ID *d* of the directly preceding node of 1.5 in document order, which is 1.3.5.3.1 in the example. 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. This strategy ensures independence from the document structure, i.e., the number of descendants between *ps* and *cn* does not matter. We have found similar search algorithms for the remaining four axes. 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. 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 [Al-Khalifa et al. 2002] adjusted to DeweyIDs, and additionally expanded to support the *preceding-sibling–following-sibling* and *preceding–following* dimensions.

Another aspect of node services to support fine-grained concurrency control for collaborative use of XML documents is of outmost importance. Although predicate locking of XQuery and XUpdate-like statements [XQuery 2004] would be powerful and elegant, its implementation rapidly leads to severe drawbacks such as the need to acquire large lock granules 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 [Haustein & Härder 2005] and empirically identified their superiority[3] [Haustein et al. 2005a].

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, [Graefe 1993]). 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 [Graefe 2000] will occur in XDBMSs, too.

2.6 Relational over XML

The issues of a simultaneous support of XML and relational database management were explored in [Halverson et al. 2004]. Questions controversially discussed so far are »Will the DBMSs of the future be hybrids, storing both relational and XML data?« 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 latter question was discussed under the name ROX: Relational over XML. 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. Mapping SQL requests onto XQuery and attaining high-performance transaction workloads as familiar from RDBMSs on native XML document trees would probably lead to a kind of »killer application«. However, it seems to be very unlikely that query evaluation efficiency and concurrency control optimization common in RDBMSs can be achieved by a system which needs additional layers for the SQL/XQuery mapping on top of those in Fig. 1.

3 Database Caching

Caching, in general, is a proven remedy to increase scalability and performance behavior of large, distributed database applications as well as to improve user-perceived latency (response time) and availability. Former approaches include static replication and full-table caching which may cause expensive consistency maintenance under certain update profiles while not providing the kind of locality support required. Therefore, adaptive caching tailored to the specific workload characteristics of an application is highly desirable.

3.1 Optimizing the Entire User-to-Data Path

As transactional Web applications (TWAs) must deliver more and more dynamic content and often updated information, Web caching should be complemented by techniques that are aware of the consistency and completeness requirements of cached data (whose source is dynamically changed in backend databases) and that, at the same time, adaptively respond to changing workloads. Because the provision of transaction-consistent and timely data is now a major concern, optimization of Web applications has to consider the entire user-to-data path (see Fig. 3). In contrast to Web caching where single Web objects are kept ready somewhere in caches in the client-to-server path, database caching is used to optimize data requests on the remaining path from application servers to the backend database. Because the essential caching issues in the path up to the Web server(s) are already addressed in

3. 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, XTC ran the 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.

sufficient detail in lots of publications [Podlipinig & Böszörmenyi 2003], we target at specific problems related to the path towards database-managed data. For this relatively new problem, currently many database vendors are developing prototype systems or are just extending their current products, e.g., [IBM DB2, Larson et al. 2004], to respond to the recently uncovered bottleneck for Web information systems or e*-applications.

What is the technical challenge of all these approaches? When user requests require responses to be assembled from static and dynamic contents somewhere in a Web cache, the dynamic portion is often generated by a remote application server, which in turn asks the backend DB server for up-to-date information, thus causing substantial latency. An obvious reaction to this performance problem is the migration of application servers to data centers closer to the users: Fig. 2 illustrates that clients select one of the replicated Web servers »close« to them in order to minimize its response time. This optimization is amplified if the associated application servers can instantly provide the expected data – frequently indicated by geographical contexts. But the displacement of application servers to the edge of the Web alone is not sufficient; conversely it would dramatically degrade the efficiency of database support because of the frequent round trips to the then remote backend DB server, e.g., by open/next/close loops of cursor-based processing via SQL application programming interfaces (APIs). As a consequence, frequently used data should be kept close to the application servers in so-called DB caches. Note, the backend DB server cannot be moved to the edge of the Web as well, because it has to serve several application servers distributed in wide-area networks. On the other hand, replication of the entire database at each application server is too expensive, because DB updates can be performed via each of them. A flexible solution should not only support database caching at mid-tier nodes of central enterprise infrastructures, but also at edge servers of content delivery networks or remote data centers.

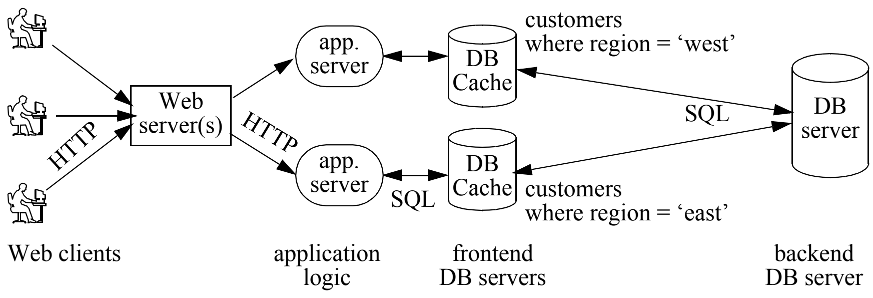

Fig. 3: The entire user-to-data path in DB-based Web applications

3.2 Objectives

Another important aspect of practical solutions is to achieve full cache transparency for applications, that is, modifications of the API are not tolerated. This application transparency is a key requirement of database caching, which also distinguishes caching from replication. It gives the cache manager the choice at run time to process a query locally or to send it to the backend DB to comply with strict consistency requirements, for instance.

The ultimate goal of database caching is to process frequently requested DB operations close to the application. Therefore, the complexity of these operations and, in turn, of the underlying data model essentially determines the required mechanisms. The use of SQL implies a considerable challenge because of its declarative and set-oriented nature. This means that, to be useful, the cache manager has to guarantee that queries can be processed in the DB cache, that is, the sets of records (of various types) satisfying the corresponding predicates – denoted as predicate extensions – must be completely in the cache. This completeness condition, the so-called predicate completeness, ensures that the query evaluation semantics is equivalent to the one provided by the backend.

A full-fledged DBMS used as cache manager offers great advantages. A substantial portion of the query processing logic (parsing, optimization, and execution) has to be made available anyway. By providing the full functionality, additional database objects such as triggers, constraints, stored procedures, or access paths can be exploited in the cache thereby simulating DB semantics locally and enhancing application performance due to increased locality. Furthermore, transactional updates seem to be conceivable in the cache (some time in the future), and, as a consequence, continued service for TWAs when backend databases become unavailable.

Note, a cache usually contains only subsets of records pertaining to a small fraction of backend tables. Its primary task is to support query processing for TWAs, which typically contain up to 3 or 4 joins [Altinel et al. 2003]. Often the number of cache tables – featuring a high degree of reference locality – is in the order of 10 or less, even if the backend DB consists of hundreds of tables.

A federated query facility as offered in [IBM DB2] allows cooperative predicate evaluation by multiple DB servers. This property is very important for cache use, because local evaluation of some (partial) predicate can be complemented by the work of the backend DB server on other (partial) predicates whose extensions are not in the cache.

3.3 Approaching a Solution for Database Caching

The conceptually most simple approach – namely, full-table caching, which replicates entire contents of selected backend tables – attracted various DB cache products [Oracle 2005a]. It seems infeasible, however, for large tables even under moderate update dynamics, because replication and maintenance costs may outweigh the potential savings on query processing.

Traditional approaches to caching at a finer granularity are settled at the object level and, hence, only support access to objects by identifiers. When the cache receives a declarative query, it is generally impossible to decide whether a complete answer can be provided without querying the backend DB. Semantic descriptions of the cached data, however, enable the cache manager to determine the completeness of query results.

So far, most approaches to DB caching were primarily based on the use of single tables, sometimes called semantic caching [Dar et al. 1996], or on materialized views and their variants [Amiri et al.

2003]. A materialized view consists of a single table whose columns correspond to the set of output attributes $O_V = \{O_1, ..., O_n\}$ and whose contents are the query result V of the related view-defining query Q_V with predicate P. Materialized views can be loaded into the DB cache in advance or can be made available on demand, for example, when a given query is processed the nth time ($n \geq 1$), exhibiting some kind of built-in locality and adaptivity mechanism. When they are used for DB caching, essentially independent tables, each representing a query result V_i of Q_{Vi}, are separately cached in the frontend DB. In general, query processing for an actual Q_A is limited to a single cache table. The result of Q_A is contained in V_i, if P_A is logically implied by P_i (subsumption) and if O_A is contained in O_{Vi}. Only in special cases, a union of cached query results, for example, $V_1 \cup V_2 \cup ... \cup V_n$, can be exploited. In contrast, a superset of the attributes Q_{Vi} may potentially enhance the caching benefit of V_i, but, on the other hand, it may increase the storage and maintenance costs.

Static methods for DB caching, where the cache contents must be prespecified and possibly loaded in advance, are not very interesting. Such approaches are sometimes called declarative caching and do not comply with challenging demands like self-administration and adaptivity[4]. Hence, what are the characteristics of a promising solution when the backend DB is (frequently) updated and the cache contents must be adjusted dynamically?

Constraint-based database caching promises a new quality for the placement of data close to their application. The key idea is to accomplish for some given types of query predicates P *predicate completeness* in the cache such that all queries eligible for P can be evaluated correctly [Härder & Bühmann 2004]. Because predicates form an intrinsic part of a data model, the various kinds of eligible predicate extensions are data-model dependent, that is, they always support only specific operations of a data model under consideration.

Suitable cache constraints for these predicates have to be specified for the cache. They enable cache loading in a constructive way and guarantee, when satisfied, the presence of their predicate extensions in the cache. The technique does not rely on the specification of static predicates: The constraints are parameterized making this specification adaptive; it is completed when the parameters are instantiated by specific values. An »instantiated constraint« then corresponds to a predicate and, when the constraint is satisfied – i.e., all related records have been loaded – it delivers correct answers to eligible queries. Note, the union of all existing predicate extensions flexibly allows the evaluation of their predicates, i.e., $P_1 \cup P_2 \cup ... \cup P_n$ or $P_1 \cap P_2 \cap ... \cap P_n$ or subsets/combinations thereof, in the cache.

There are no or only simple decidability problems whether predicates can be evaluated. Only a simple probe query is required at run time to determine the availability of eligible predicate extensions. Furthermore, because all columns of the corresponding backend tables are kept, all *project operations* possible in the backend DB can also be performed in the cache. Other operations like *selection and join* depend on specific *cache constraints*. Since full DB functionality is available, the results of these queries can further be *refined* by selection predicates such as Like, Null, etc. as well as processing options like Distinct, Group-by, Having (potentially restricted), or Order-by.

Moreover, 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 [Härder & Bühmann 2004].

On the other hand, handling of updates is a critical problem and could be alleviated by applying different update models to DB caching. Instead of processing all (transactional) updates in the backend DB first, one could perform them in the cache (under ACID protection) 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.

As compared to the most sophisticated client/server DBMSs – the query server approach [Härder & Rahm 2001] – the situation is here even more challenging. While locality preservation in the (client-side) query result buffer of a query server can take advantage of application hints [Deßloch et al. 1998], adaptivity of database caching is a major challenge for future research [Altinel et al. 2003]. 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 [Guo et al. 2004].

4 The Next Database Revolution Ahead?

So far, we could show that the *invariants in database management* observed in relational DBMSs also determine the mapping steps of an XDBMS architecture, although we had to refine and adjust the layers and algorithms to the fine-grained and record-oriented tree structures of XML documents. Fig. 4 shows simplified three-layer architectural models for relational and XML DBMSs thereby contrasting the difference of these record-oriented architectures to those of other data types.

Progress is made for some of these data types. Thanks to the object-relational database development, data and procedures are now being joined. Decades of discussion about the inside-the-database/outside-the-database dichotomy of application code are over: »The Java or common language runtimes have been married to relational database engines so that the traditional EJB-SQL outside-inside split has been eliminated. Now Beans or business logic can run inside the database« [Gray 2005]. Indeed, the most recent generation of object-oriented environments provides a common runtime capable of supporting good performance for nearly all languages, in particular, Java and C#. These languages have also been fully integrated into some object-relational databases.[5] Hence, databases

4. Minimum interaction by the database administrator is desirable when a large number of caches exists, e.g., Akamai's network has nearly 15,000 edge caching servers [Akamai].

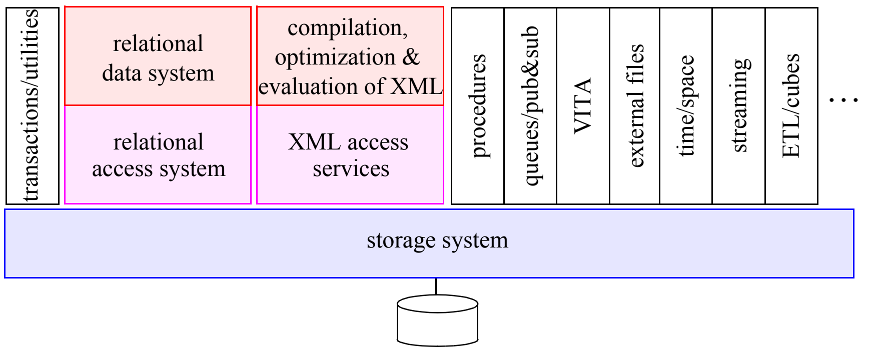

Fig. 4: Desirable extensions for future DBMS architectures

also have the opportunity to become the preferred integration vehicle for application development environments. With these integration efforts, fields become objects (values or references), records become vectors of objects (fields), and tables become sequences of record objects. As a consequence, databases can be perceived as collections of tables (of objects). According to Jim Gray, this objectified view of database systems embodies a quantum leap for revolutionary developments for other data types. On the other hand, such a development would stand for the re-rise of the object-relational DBMS concepts.

With these concepts in mind, the integration of persistent queues is only a little step, because – based on the available ACID properties – we can implement tailored queuing semantics using, for example, stored procedures (and triggers). Based on such persistent queues [DB2 2005, Oracle 2005b], the DB middleware can provide message brokering and pub&sub services. Because messages are data, too, we could directly exploit enhanced XDBMSs to enable native storage and management of messages in XML format. 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? Because the new data types can often reuse only the external storage mapping, specialized higher-level layers have to be implemented for each of them. For example, VITA types managed in tailored DB buffers are typically delivered (in variable-length junks) directly 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.

Often it makes not much sense to store and manage all data in a DBMS. For data that must be »streamed« to the application within a specified time period in order to be meaningful, e.g., video frames or audio, the use of specialized file servers optimized for delivery of such data would be more appropriate. Locating the data close to the application further improves application performance. To loosely integrate (huge numbers of) such external files, the DataLink concept provides a framework to control referential integrity, access control, and recoverability for external data by a DBMS [Bhattacharya et al. 2002, Hsiao & Narang 2000, Melton et al. 2001].

Fig. 4 illustrates some characteristics of future DBMS architectures: layer models of different kinds, architectural models for specific data types, integration of external files, etc. Often the OS services or, at best, the storage system represent the least common denominator for these desired DBMS extensions. For many of these future DBMS extensions, systematic architectural approaches are not known today. Therefore, we rather have to deviate to vague speculations.

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 [Gray 2004] for which the metaphor of a database ecosystem is used. As a consequence, we obtain a collection of heterogeneous DBMSs (and file systems) which have to be made accessible for the applications – as transparently as possible – by suitable APIs. Apparently, this database ecosystem enables a large degree of scalability. Furthermore, 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« [Gray 2004]. And because databases should be accessible from anywhere, Web services may become the means of choice by which we federate heterogeneous database systems. Furthermore, because these systems cooperate on behalf of applications, ACID protection has to be assured for all messages and data taking part in a transaction [Gray & Reuter 1993]. Applications and the corresponding transactions currently running locally under control of a single DBMS will then typically access a database ecosystem and, as a consequence, will turn into distributed and heterogeneous processing compounds needing federation services and 2PC protocols.

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[6] – both leading to contradicting guidelines for the architectural design.

5 Dependable Adaptive Information Systems

Along these lines, we have to broaden our perspective and deviate from the unilateral focus on the DBMS engine alone. Many parts and functions of modern societies heavily depend on information systems serving a wide spectrum of uses in administration, production, trading, banking, traffic control, tele-communication, science, law-enforcement, etc. Hence, we have to consider a bigger pic-

5. For OLTP, this opportunity could mean that databases encapsulate business logic using stored procedures. Then the TP-lite discussion is on the stage back again where three-tier client/server systems running under control of a TP monitor (TP-heavy) are challenged by two-tier architectures [Gray 2005].

6. »Indeed, if every file system, every disk, every phone, every TV, every camera, and every piece of smart dust is to have a database inside, then those database systems will need to be self-managing, self-organizing, and self-healing« [Gray 2005].

ture including the information system perspective as an ecosystem consisting of DBMS engine(s), DB-related middleware, and application integration.

5.1 Design Objectives

One of the Grand Challenges in trustworthy computing (stated by the Computing Research Association) is: »Build Systems You Can Count On«. For the realm of information systems, Jim Gray has pinpointed the consequences by (at least) three of the dozen long-term research & development goals stated in his 1999 Turing Lecture:

- **Trouble-Free**
  Build a system used by millions of people each day and yet administered and managed by a single part-time person.
- **Secure**
  Assure that the system (solving previous problem) only services authorized users, service cannot be denied by unauthorized users, and information cannot be stolen (and prove it).
- **Always-Up**
  Assure that the system is unavailable for less than one second per hundred years – eight 9s of availability (and prove it).

By solving these problems, we accomplish dependable adaptive information systems (DAISs). When designing such information systems, software correctness (the information system meets its specification) is only one among several dependability issues. To provide for high availability and to protect against data loss, recovery-oriented computing (ROC) is of significant importance. Various kinds of failures are unavoidable (transaction abort, system crash, media failure) and have no solution to be achieved by software alone. DBMSs have a long history of successful ROC and guarantee, in any failure case, a transaction-consistent state of the data. In contrast, adaptivity is a far less common property of DBMSs or even information systems. A DBMS should automatically adapt to varying numbers of users, to capacity changes in resources, to the unavailability of data sources, and to different response-time/throughput requirements of the incoming transactions. Optimized service under workload/usage changes and scalability (Web information systems may attract very large numbers of concurrent users) are still design goals hard to achieve. Finally, dependability requires high degrees of system availability and robustness, tolerance against deliberate attacks, as well as provision for trust management and security to an extent so far unknown even in centralized DBMSs.

Future generation information systems are no longer built around a central DBMS. For their applications, they must provide access to several databases, coordination of the concurrent accesses, use of other services with unknown internal structure, and support of business processes by long and structured transactions, often designed as workflows. In particular, these information systems have to cope with the following requirements:

- Wide-area distribution
- Openness of system structure and autonomy of components
- Heterogeneity of structure and content
- Long-term interactive processes.

The increase in power, accessibility, and flexibility provided by such information systems comes at a high price. Distribution, openness, and component autonomy lead to less control over the system and to new possibilities for faults and deliberate attacks – with negative consequences for dependability. A future DAIS, for example, integrating data sources from the Web, will necessarily have to cope with different kinds of heterogeneity and dynamic information integration. For this reason, research currently takes a strong focus on XML technology to transparently query data from different kinds of data sources thereby »homogenizing« the query result to a unified view. However, such approaches can only partially solve the semantic heterogeneity problem at the structure level (schema level). At the content level (instance level), the resulting problems are rather elusive and far away from an adequate solution. The entirety of these requirements also makes it more difficult to develop information systems that automatically adapt to changes in resource availability and that are scalable to the needs of different application scenarios. For example, long-term processes lead to complex dependencies between the applications and the persistent data repositories and their subcomponents. In such distributed component systems, resource planning and adaptation require state and control information delivered by distributed observations that go beyond single components.

5.2 Architectural and Transactional Model

Because of the new kinds of component interactions, information exchanges, and mutual dependencies, the design of the system architecture and middleware becomes at least equally important as the development/layout of the individual components and their algorithms for the higher-ranking objectives of adaptivity and dependability. For these reasons, the challenge to build dependable and adaptive information systems of the future is particularly concerned with architectural issues of software systems, component models, and new concepts of cooperation and synchronization. An important problem is to tame the architecture, its flexibility and complexity to a degree that still enables extensibility (permanent change of technology and functional growth) and adaptivity without losing dependability. To construct such DAISs, these three conflicting design goals have to be considered together.

Although these developments are important prerequisites to implement DAISs, they are by far not sufficient to reach the fundamental goals. A particular challenge for failure-free component interaction and cooperation is that their context models often do not exactly match. Furthermore, successful adaptation can only be achieved based on observation and prediction. If the application processes rely on a specific execution model that is known by the other participating components of the information system, these components can use the knowledge of the execution model to plan their actions. If they have no information about the application behavior, reliable adaptation is not possible. Similar to the transaction concept which indicates the intent to perform a number of DB operations in an atomic way, we need equivalent, but more flexible execution models that allow to convey properties of the upcoming execution to the information system component responsible for it.

For these and other aspects, a number of models and techniques have already been developed. For example, (closed) nested transactions enable a tree-like structuring of executions thereby enabling intra-transaction parallelism and fine-grained recovery while preserving ACID for the entire transaction tree [Weikum & Vossen 2002]. In a DAIS, open nested transactions have to be adjusted to the distributed and heterogeneous environment thereby aggravating the compensation problems in case of a failure. When autonomous systems are involved, so-called agreement protocols have to be introduced to support secure distributed computing in group cooperation between applications. Behavioral specification techniques for component interfaces are available and can be enriched by non-functional properties. However, it is not yet clear how these models and techniques evolve and work together. Therefore, we should refine these concepts and integrate their properties in a well-defined semantic framework that provides the foundation for adaptivity and dependability while the participating DBMSs remain extensible.

5.3 Adaptivity in a DAIS

So far, information hiding and layers as abstract machines were the cornerstones for the design of large evolutionary systems and, in particular, of the DBMS engine. The new design goal of enhanced adaptivity, however, challenges these proven concepts. Typically, adaptable component behavior cannot be achieved by exploiting local »self«-observations (knowledge) alone. Hence, autonomic computing principles applied to DBMS components require more information exchange across components (introducing additional dependencies) to gain a more system-oriented view when decisions relevant for behavioral adaptations have to be made.

DBMS Component Adaptivity

In this context, components are large-grained rather than fine-grained objects, that is, storage manager, query optimizer, etc. as examples. Heuristic mechanisms (rules of thumb) will sometimes provide adequate static settings for specific tuning knobs, but will quickly exhaust in more complex situations (e.g., when the cache size has to be adjusted to varying workloads). As a consequence, typical component adaptations often require the collection of statistical data and imply a continuous adjustment of tuning or behavioral parameters. They may be as complex as a learning optimizer [Markl et al. 2003].

The provision of inter-component adaptivity is more challenging, because tight cooperation among separate system components is necessary where each component is aware of only its optimal local planning situation or decision. Some examples may illustrate this issue. Dynamic query evaluation plans require alternate plans for different load situations and resource availability. In contrast, the query optimizer could ask the index manager to dynamically create a new index on a table – a decision concerning the physical DB which could quickly be amortized when the future query load evolves as anticipated by the optimizer. Such a decision again is an inter-component optimization problem. Optimized dynamic index selection, i.e., whether a new index for a table is cost-effective, needs the cooperation of several components to collect statistical data which identify potential costs and savings caused by new indexes, before a »performance planner« decides that such an adaptation measure is beneficial for the entire system behavior. Or the resolution of overload conflicts due to lock contention can only be handled by a cooperation of the lock manager and load balancer. The general solution to achieve »self-*« system properties in such situations consists of the four phases observation, analysis, planning, and reaction which form an online feedback control loop with which (hopefully) optimal situation-specific system decisions can be derived [Weikum et al. 2002].

System-level Adaptivity

Complex, long-living workflows are the most important mechanism to implement business processes in DAISs. Because each workflow instance can be considered as a persistent, recoverable object and typically has a large amount of state information, again DBMSs come into play to provide stable states of workflow instances and to take over quite a share of their functionality. Adaptation at the system level primarily means to allow for flexible workflow schemas, which may evolve responding to unforeseen situations (new cooperation partner, inaccessibility of data sources, unavailability of resources) or which need a sudden rescheduling. Because »the human is in the loop«, interaction timing is hard to preplan (think time). As a consequence, ACID transactions are inappropriate as control structures. Rather, open nested transactions or agreement protocols have to be used to achieve an adequate processing quality (still to be defined).

5.4 Adaptivity versus Dependability

Trouble-free operation of a DAIS 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 [Gray 2003]. Online feedback control loops are key to achieve such adaptation and »self-*« system properties, which, however, amplify the information channels across system layers. In addition, close cooperation among the participating (autonomous and heterogeneous) systems introduces many new dependencies. Too many information channels, however, 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. To develop such »always-up + secure + trouble-free« 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.

Prerequisites of Dependability

Dependability requires precaution measures for all expected failure types, which, in the first place, means tailored ROC mechanisms, i.e., logging and recovery algorithms optimized w.r.t. the critical system demands (e.g., high availability). Such issues are amenable to known engineering principles and can be proven by checking the resulting performance behavior. However, many other aspects such as security or trust are non-functional and cannot easily be »attached« to a specific component, let alone optimized by using mathematical models.

For adaptivity, self-observation and self-analysis primarily aim at the automatic system accommodation and optimization under changes in user behaviors and workloads. Future DAISs and, in particular, DBMSs typically operate in an unattended way. In contrast, improving the dependability aspect asks for automatic recognition of internal malfunctions and malfunction of communicating components, identification of data corruption, intrusion detection, application failures, etc. In other words, such capabilities require making the information system more self-aware. To automatically recognize deviations from the expected behavior, the system needs explicit models of its behavior and the surrounding system/environment in which it participates/cooperates – these may be guided by default policies to proceed in unexpected situations and by personal profiles summarizing the user needs to cope with extraordinary user requests/attacks [Lowell 2003].

5.5 Towards Dependability Engineering

How to improve dependability properties? All the issues sketched so far call for even more information system functionality which opens an immense chasm resulting 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.* Therefore, we need to develop a kind of dependable systems' engineering which should cover all dependability aspects in a system model (enclosing the DBMS architecture). Abstract principles for such an idea include [Weikum et al. 2002]:

- a highly componentized system structure with
- components of limited and relatively simple functionality (similar to RISC-style designs) and
- well-controlled component interfaces as narrow as possible to keep the interaction complexity between components to a minimum.

At the moment, it is not clear how such a design can be accomplished and how it affects other system objectives, in particular, performance. It will be a great challenge to define dependability in an operational way, such that the dependability state of a system can be evaluated and that the success of additional measures towards a higher degree of dependability can be captured in a quantitative way. However, currently the bad news is that the state-of-the-art architectural designs of universal DBMSs are against the spirit of a dependability engineering.

Nevertheless, a future architecture has to observe the implications of autonomic and trouble-free computing even under various forms of pressures and deliberate attacks, that is, it has not only to be guided by data independence and evolution, but has to be developed along the lines of adaptivity and dependability.

5.6 More Challenges Ahead

Another of the Grand Challenges in trustworthy computing is: »Within 10 years, develop quantitative information-systems risk management that is at least as good as quantitative financial risk management.« Currently, we have made many observations and experiences, but we do not understand the full nature of what causes IT risk and the emergent behavior of some vulnerabilities and systems. Furthermore, we assume in our assessments and countermeasures the independence and single-failure-at-a-time properties, although failures in networked systems are not independent. To approach a solution for DAISs, mathematical modeling where models for operational risks are explored, is helpful. On the other hand, progress towards more intelligent adaptive information system interfaces may facilitate trust management and effective defense of, for example, hacker attacks or accidental misuse of the DAIS. Filtering and assessment of user requests help to achieve better usability (personalization and context awareness) and may prevent »dangerous« situations which would possibly lead to system failures, unavailability of resources, denial of service, and so on.

6 Conclusions

In this paper, we primarily considered architectural issues of database research and development which may happen in the next decade of the BTW – the German conference »Database Systems in Business, Technology, and the Web«. We observed that the DBMS architecture for the declarative and set-oriented (relational) processing paradigm of record-like structures embodied by the five-layer hierarchical model cannot accommodate all the desired requirements and extensions for new and future data management scenarios.

The more the integration of new data types requires new processing invariants, the less the original architecture is able to cope with them. It is an open question how the architecture for an e*xtensible object-relational system where non-procedural relational operators manipulate object sets* will evolve in detail. Furthermore, a future architecture has to observe the implications of autonomic and trouble-free computing even under various forms of »pressures« and deliberate attacks, that is, it has to be guided by data independence and evolution and, at the same time, developed along the lines of adaptivity and dependability. At any rate, the good news for all DB researchers is that there are plenty of challenges still ahead in the area of DBMS architecture and that new ones evolve as soon as we seriously cope with these challenges.

Acknowledgements. Discussions with Andreas Bühmann, Stefan Deßloch, Jernej Kovse, Christian Mathis, Bernhard Mitschang and Joachim Thomas who also carefully read a preliminary version helped to shape the paper and to improve its final version.

References

[Akamai] Akamai Technologies Inc.: Akamai EdgeSuite. http://www.akamai.com/en/html/services/edgesuite.html

[Al-Khalifa et al. 2002] *Al-Khalifa, S.; Jagadish, H. V.; Patel, J. M.; Wu, Y.; Koudas, N.; Srivastava, D.:* Structural Joins: A Primitive for Efficient XML Query Pattern Matching. Proc. 18th Int. Conf. on Data Engineering, 141 (2002)

[Altinel et al. 2003] *Altinel, M.; Bornhövd, C.; Krishnamurthy, S.; Mohan, C.; Pirahesh, H.; Reinwald, B.:* Cache Tables: Paving the Way for an Adaptive Database Cache. VLDB 2003: 718-729

[Amiri et al. 2003] *Amiri, K.; Park, S.; Tewari, R.; Padmanabhan, S.:* DBProxy: A Dynamic Data Cache for Web Applications. ICDE Conference 2003: 821–831

[Bhattacharya et al. 2002] *Bhattacharya, S., Mohan, C., Brannon, K., Narang, I., Hsiao, H., Subramanian, M.:* Coordinating backup/recovery and data consistency between database and file systems. SIGMOD Conference 2002: 500-511

[Dar et al. 1996] *Dar, S., Franklin, M., Jónsson, B., Srivastava, D., Tan, M.:* Semantic Data Caching and Replacement. VLDB 1996: 330–341

[DB2 2005] Administration of file and queue management on DB2, http://publib.boulder.ibm.com/infocenter/txen/index.jsp?topic=/com.ibm.txseries510.doc/erzhab0028.htm

[Deßloch et al. 1998] *Deßloch, S.; Härder, T.; Mattos, N. M.; Mitschang, B.; Thomas, J.:* Advanced Data Processing in KRISYS: Modeling Concepts, Implementation Techniques, and Client/Server Issues. VLDB J. 7(2): 79-95 (1998)

[DOM 2004] Document Object Model (DOM) Level 2 / Level 3 Core Specification, W3C Recommendation (Nov. 2000 / Apr. 2004)

[Florescu & Kossmann 1999] *Florescu, D.; Kossmann, D.:* Storing and Querying XML Data using an RDMBS. IEEE Data Eng. Bull. 22(3): 27-34 (1999)

[Graefe 1993] *Graefe, G.:* Query Evaluation Techniques for Large Databases. ACM Comput. Surv. 25(2): 73-170 (1993)

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

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

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

[Gray 2005] *Gray, J.:* A Call to Arms. ACM Queue 3:3, 30-38, April 2005

[Gray & Reuter 1993] *Gray, J.; Reuter, A.:* Transaction Processing: Concepts and Techniques. Morgan Kaufmann 1993

[Guo et al. 2004] *Guo, H.; Larson, P.-A.; Ramakrishnan, R.; Goldstein, J.:* Relaxed Currency and Consistency: How to Say »Good Enough« in SQL. SIGMOD Conference 2004: 815-826

[Härder 2005a] *Härder, T.:* DBMS Architecture – Still an Open Problem. BTW, LNI P-65, Springer, 2-28, 2005

[Härder 2005b] *Härder, T.:* XML Databases – Plenty of Architectural Problems Ahead. ADBIS Conference 2005 (keynote paper), LNCS, Springer

[Härder 2005c] *Härder, T.:* DBMS Architecture – The Layer Model and its Evolution. Datenbank-Spektrum, Heft 13/2005, 45-57

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

[Haustein 2005] *Haustein, M.*: Eine XML-Programmierschnittstelle zur transaktionsgeschützten Kombination von DOM, SAX und XQuery, BTW, LNI P-65, Springer, 265-284, 2005

[Haustein & Härder 2005] *Haustein, M.; Härder, T.:* Optimizing Concurrent XML Processing, submitted (2005), http://wwwdvs.informatik.uni-kl.de/pubs/p2005.html

[Haustein & al. 2005a] *Haustein, M.; Härder, T.; Luttenberger, K.:* Contest of Lock Protocols, submitted (2005), http://wwwdvs.informatik.uni-kl.de/pubs/p2005.htm

[Haustein & al. 2005b] *Haustein, M.; Härder, T.; Mathis, C.; Wagner, M.:* DeweyIDs—The Key to Fine-Grained Management of XML Documents, appears in: Proc. 20th SBBD, Uberlandia, 2005), http://wwwdvs.informatik.uni-kl.de/pubs/p2005.html

[Härder & Rahm 2001] *Härder, T.; Rahm, E.*: Datenbanksysteme: Konzepte und Techniken der Implementierung, 2nd edition, Springer 2001

[Halverson et al. 2004] *Halverson, A.; Josifovski, V.; Lohman, G.; Pirahesh, H.; Mörschel, M.:* ROX: Relational Over XML. VLDB 2004: 264-275

[Hsiao & Narang 2000] *Hsiao, H., Narang, I.:* DLFM: A Transactional Resource Manager. SIGMOD Conference 2000: 518-528

[IBM DB2] IBM DB2 Universal Database (V 8.2). http://www.ibm.com/software/data/db2/

[Jagadish et al. 2002] *Jagadish, H. V., Al-Khalifa, S., Chapman, A., Lakshmanan, L. V. S., Nierman, A., Paparizos, S., Patel, J. M., Srivastava, D., Wiwatwattana, N., Wu, Y., Yu, C.:* TIMBER: A native XML database. VLDB J. 11(4): 274-291 (2002)

[Larson et al. 2004] *Larson, P.-Å.; Goldstein, J.; Zhou, J.:* MTCache: Mid-Tier Database Caching in SQL Server. ICDE 2004: 177-189

[Lowell 2003] *Lowell Workshop Summary:* The Lowell Database Research Self Assessment. The Computing Research Repository, CoRR cs.DB/0310006 (2003)

[Markl et al. 2003] *Markl, V., Lohman, G. M., Raman, V.:* LEO: An autonomic query optimizer for DB2. IBM Systems Journal 42(1): 98-106 (2003)

[Melton et al. 2001] *Melton, J., Michels, J.-E., Josifovski, V., Kulkarni, K. G., Schwarz, P. M., Zeidenstein, K.:* SQL and Management of External Data. SIGMOD Record 30(1): 70-77 (2001)

[O'Neil et al. 2004] *O'Neil, E. J.; O'Neil, P. E.; Pal, S.; Cseri, I.; Schaller, G.; Westbury, N.:* ORDPATHs : Insert-Friendly XML Node Labels. Proc. SIGMOD 2004, 903-908 (2004)

[Oracle 2005a] *Oracle Corporation:* Internet Application Sever Documentation Library, http://otn.oracle.com/documentation/appserver10g.html

[Oracle 2005b] Advanced Queuing In Oracle9i, http://www.oracle-base.com/articles/9i/AdvancedQueuing9i.php

[Podlipinig & Böszörmenyi 2003] *Podlipinig, S.; Böszörmenyi, L.:* A Survey of Web Cache Replacement Strategies. ACM Computing Surveys 35:4, 374–398 (2003)

[Rahm 1994] *Rahm, E.:* Mehrrechner-Datenbanksysteme - Grundlagen der verteilten und parallelen Datenbankverarbeitung. Addison-Wesley 1994

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

[Weikum & Vossen 2002] *Weikum, G.; Vossen, G.:* Transactional Information Systems – Theory, Algorithms, and the Practice of Concurrency Control and Recovery, Morgan Kaufmann 2002[7]

[XQuery 2004] XQuery 1.0: An XML Query Language. W3C Working Draft (Oct. 2004)

Theo Härder obtained his Ph.D. degree in Computer Science from the TU Darmstadt in 1975. In 1976, he spent a postdoctoral year at the IBM research Lab in San Jose and joined the project System R. In 1978, he was associate professor for Computer Science at the TU Darmstadt. As a full professor, he is leading the research group DBIS at the TU Kaiserslautern since 1980. He is the recipient of the Konrad Zuse Medal (2001) and the Alwin Walther Medal (2004) and obtained the Honorary Doctoral Degree from the Computer Science Dept. of the University of Oldenburg in 2002.

Theo Härder's research interests are in all areas of database and information systems - in particular, DBMS architecture, transaction systems, information integration, and Web information systems. He is author/coauthor of 7 textbooks and of more than 200 scientific contributions with >100 peer-reviewed conference papers and >50 journal publications. His professional services include numerous positions as chairman of the GI-Fachbereich »Databases and Information Systems«, conference/program chairs and program committee member, editor-in-chief of Informatik – Forschung und Entwicklung (Springer), associate editor of Information Systems (Elsevier), World Wide Web (Kluver), and Transactions on Database Systems (ACM). He serves as a DFG expert and is the Chairman of the Selection Committee for Computer Science of the Bavarian Network of Excellence (Elitenetzwerk Bayern). He was chairman of the Center for Computed-based Engineering Systems at the University of Kaiserslautern, member of two DFG SFBs (124, 501), and co-coordinator of the National DFG Research Program »Object Bases for Experts«.

Prof. Dr.-Ing. Dr. h.c. Theo Härder
Technische Universität Kaiserslautern
Fachbereich Informatik
Postfach 3049
67653 Kaiserslautern
haerder@informatik.uni-kl.de
http://www.haerder.de

7. Many references are copied from http://www.informatik.uni-trier.de/~ley/db/index.html. This support of Michael Ley is greatly appreciated.