DBMS Architecture—Still an Open Problem

Theo Härder

Database and Information Systems
Dept. of Computer Science
University of Kaiserslautern, Germany

Abstract. More than two decades ago, DB researchers faced up to the question of how to design a data-independent database management system (DBMS), that is, a DBMS which offers an appropriate application programming interface (API) to the user and whose architecture is open for permanent evolution. For this purpose, 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. We review the basic concepts and implementation techniques of this model and survey the major improvements achieved in the system layers to date. Furthermore, we consider the interplay of the layered model with the transactional ACID properties and again outline the progress obtained. In the course of the last 20 years, this DBMS architecture was challenged by a variety of new requirements and changes as far as processing environments, data types, functional extensions, heterogeneity, autonomy, scalability, etc. are concerned. We identify the cases which can be adjusted by our standard system model and which need major extensions or other types of system models.

1 Introduction

In the seventies, the scientific discussion in the database (DB) area was dominated by heavy arguments concerning the most suitable data model, sometimes called a religious war. It essentially focussed on the question of which abstraction level is appropriate for a DB application programmer. The network data model seems to be best characterized by "the more complex the pointer-based data structure, the more accurate is the mini-world representation". However, it offers only very simple operations forcing the programmer to navigate through cursor-controlled data spaces. In contrast, a "data structure of spartan simplicity" (E. F. Codd) and value-based relationship representation are provided by the relational data model. Each additional embellishment (modeling option) needs additional operations and thus leads to greater model complexity. Because the result of every DB operation is a table (a multi-set of an unnamed type in SQL), it offers the closure property which was a prerequisite for its practically very significant concept of *views*. In that time, the decision concerning the most appropriate data model could be pinpointed to "record orientation and pointer-based, navigational use" vs. "set orientation and value-based, declarative use". Far ahead of the common belief of his time, E. F. Codd taught us that simplicity is the secret of data independence—a property of the data model and the database

management system (DBMS) implementing it. A high degree of data independence is urgently needed to let a system “survive” the permanent change in computer science in general and in the DB area in particular.

How should we design a DBMS whose architecture is open for permanent change or evolution? More than two decades ago, this was already a difficult question, because nobody knew what “permanent change” means. On the other hand, it was rather simple compared to the time being, because we only thought about storing and managing objects of the relational or network (or hierarchical) data models, that is, sets of simply structured records or tables. Nowadays, however, important DBMS requirements include data streams, unstructured or semi-structured documents, time series, spatial objects, and so on. What were the recommendations to achieve the system properties for which the terms *physical* and *logical data independence* were coined?

It is immediately clear that a monolithic approach to DBMS implementation is not very reasonable. It would mean to map the data model functionality (e.g., SQL) in a single step to the interfaces offered by external storage devices, e.g., read/write block. As we have experienced during the last decades, DBMSs have a lifetime >20 or even >30 years. In that time, system evolution requirements were 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 had far-reaching consequences on storage devices, memory, connectivity (e.g., Web), and so on. But what are the guidelines of system development in such a situation to be anticipated?

The ideas of structured programming and information hiding were cornerstones guiding the way of development. “The Goto Statement Considered Harmful” (E. W. Dijkstra [13]) was translated into the DB world as the battle cry “Pointers are the Evil!”. On the other hand, the concept of *information hiding* introduced by D. L. Parnas [46] was widely accepted in academic circles as a software design principle. However, most industrial software developers did not apply the idea and many considered it unrealistic.[1] Parnas convinced us of the advantages of the *use relation*[2] [47] to be applied to hierarchical layers of large software systems. Developing a hierarchically structured system offers the following important benefits:

- The implementation of higher-level system components is simplified by the usage of lower-level system components.
- Lower-level system components are independent of functionality and modifications in higher-level system components.
- Testing of lower-level system components is possible, before the higher system levels are put into use.

---

1 taken from a Parnas’ talk *Modularization by Information Hiding*: The napkin of doom—Compiler and database experts have a lunch.They exchange a control block format on a napkin. Napkin is punched, copied, and filed. Format changes, but napkin does not. Components are coupled and don‘t work. They had to do something. I did not know what they should have done. (www.sdm.de/download/sdm-konf2001/f_3_parnas.pdf)

2 “Module A uses module B, if A calls B and the complete execution of A requires the correct execution of B.”

The resulting abstraction hierarchy hides some properties of a system level (an abstract machine) from higher-layer machines. Furthermore, the implementation of higher-level operations extends the functionality of an abstract machine. System evolution is often restricted to the internals of such abstract machines when, for example, a function implementation is replaced by a more efficient one. In case new functionality extends their interfaces, the invocation of these operations implies “external” changes which are, however, limited to the next higher layer.

But how can these concepts and objectives be accomplished? Although it is fundamentally true that “systematic abstraction is the prime task of computer science” (H. Wedekind), it is always very hard to translate the appropriate abstraction into a system structure. Did this multi-level information-hiding approach fulfil the far-reaching expectations during the last 20 years we try to look back to?

In section 2, we review the concepts of the five-layer hierarchical DBMS model and use it as an explanation model for run-time aspects, binding and information-flow dependencies. Furthermore, we survey the major improvements achieved in the various layers in the course of the past two decades. In section 3, we consider the interplay of the layered model with the transactional ACID properties and again outline the progress achieved. Section 4 discusses a variety of DBMS architectures where our layered model can be used to describe the mapping abstractions and to gain better insight into the operations modeled. In section 5, 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, before we briefly summarize our findings in section 6.

2 Hierarchical DBMS Architecture

2.1 Static Engine Architecture

Given the arguments introduced so far, it was clear to select a multi-layered hierarchical architecture to implement a centralized DBMS. However, when starting with a concrete design, many questions with no obvious answer emerged. How many layers are adequate for the entire DB-mapping process? What is an appropriate decomposition of DBMS functionality into individual layers? Do we have to provide auxiliary mapping data or meta-data for each layer separately or is a centralized meta-data repository more appropriate? Where do we allocate the transaction functionality, i.e., the ACID properties? And many questions more...

About 20 years ago, Andreas Reuter and the author proposed a mapping model or reference architecture consisting of five layers [25, 27] depicted in Table 1. 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 mean-

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.

ingful 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 a declarative data access via SQL.

The bottom layer, called *File Management,* operates on the bit pattern stored on some external, non-volatile storage device. Often in collaboration with the operating system's file management, this layer copes with the physical characteristics of each type of storage device. *Propagation Control* as the next higher layer introduces different types of pages which are fixed-length partitions of a linear address space and mapped into physical blocks which are, in turn, stored on external devices by the file management. The strict distinction between pages and blocks offers additional degrees of freedom for the propagation of modified pages. For example, a page can be stored in different blocks during its lifetime in the database thereby enabling atomic propagation schemes (supporting failure recovery based on logical logging, see section 3.4.1). To effectively reduce the physical I/O, this layer provides for a (large) DB buffer which acts as a page-oriented interface (with fix/unfix operations) to the fraction of the DB currently resident in memory.

The *Record and Access Path Management* implements mapping functions much more complicated than those provided by the two subordinate layers. For performance reasons, the partitioning of data into segments and pages is still visible at this layer. It has to provide clustering facilities and maintain all physical object representations, that is, data records, fields, etc. as well as access path structures, such as B-trees, and internal catalog information. It typically offers a variety of access paths of different types to the navigational access layer. Especially with the clustering options and the provision of flexibly usable access paths that are tailored to the anticipated workloads, this layer plays a key role for the entire DBMS performance.

The *Navigational Access Layer* maps physical objects to their logical representations and vice versa. At this interface, the user (or the modules at the next higher layer) navigates through a hierarchy or network of logical records or along logical access paths using scans of various types. A special ability is to dynamically order (sort) sets of records to support higher operations such as sort/merge joins. Finally, the *Non-procedural Access Layer* provides logical data structures (such as tables and views) with declarative operations or a non-procedural interface to the database. At the API, it provides an access-path-independent data model with descriptive languages (e.g., SQL).

Each layer needs a number of auxiliary data structures for mapping higher-level objects to more elementary ones. To indicate the type of data, Table 1 characterizes some of them.

2.2 Dynamics of Query Execution

To gain a more detailed insight into the internal DBMS tasks and dynamics, we sketch important translation, optimization, and execution steps exemplified by the following SQL query Q1:

Select B.Title, B.P-Year, B.Publisher, A.Name
From Books B, Authors A
Where B.AuthId = A.AuthId *And* A.Name = 'S*' *And* B.Subject = 'DBMS'

It is highly desirable to relieve run time from all preparation aspects of query processing and to shift as much of it as possible to query compilation time. Therefore, a particular design objective is to generate for each query a so-called access module (incorporating the query evaluation plan (QEP)), the operations of which can be directly invoked at the L4 interface (see Fig. 1). Missing (suitable) indexes on *Books* and *Authors* would enforce table (segment) scans on them in our example. Therefore, let us assume that indexes $I_{Authors}$(Name) and I_{Books}(Subject) exist and that, in turn, the optimizer plans a sort/merge join for the query evaluation. When using I-scans on the given indexes (with start and stop conditions) for the selection operations, the required sort orders on *AuthId* do not come for free. Hence, sorted objects have to be created explicitly in L4, before the join can be processed. In turn, the I-scans, which deliver the records to be sorted, fetch them via physical access paths and storage structures managed by L3[3]. The resulting access module is illustrated in Fig. 1. This module directly returns the result set of Q1. Normally, L3 offers a variety of storage structures which physically embody the indexes or other types of access paths. The most prominent ones are, of course, the "ubiquitous" B-trees or B*-trees [9] which grant rapid direct and, equally important, sorted sequential access to the indexed data. Because they are search trees, they can support range queries or may give the index sort order to the subsequent QEP operator for free.

The functionality provided by L3 needs to refer to the physical data of the DB. For this reason, the DB buffer (as part of L2) acts as the memory interface to the DB on external

[3] For example, a list prefetch operator is provided by DB2 to optimize such situations.

```
Open Scan (I_Books(Subject), Subject = 'DBMS', Subject > 'DBMS')          /* SCB1 */
Sort Access (SCB1) ASC AuthId Into T1 (AuthId, Title, P-Year, Publisher)
Close Scan (SCB1)
Open Scan (I_Authors(Name), Name >= 'S', Name > 'S')                      /* SCB2 */
Sort Access (SCB2) ASC AuthId Into T2 (AuthId, Name)
Close Scan (SCB2)
Open Scan (T1, BOF, EOF)                                                  /* SCB3 */
Open Scan (T2, BOF, EOF)                                                  /* SCB4 */
While Not Finished
Do
   Fetch Tuple (SCB3, Next, None)
   Fetch Tuple (SCB4, Next, None)
   . . .
End
```

Fig. 1 Access module for query Q1

devices and provides access to pages based on explicit request and release (so-called logical page references). Its prime objective is to exploit the (space and time) locality of DBMS processing (inter- and intra-transaction locality of page references) to minimize the number of physical page references (usually disk accesses). Because DB buffer space as well as DB volumes roughly grow at the same pace, the size ratio buffer/DB (memory/storage ratio) remains constant over time—often a ratio of 1:1000 is a good estimate. Hence, random access to DB pages and random page replacement in the buffer would result in a hit ratio of 0.1%. Fortunately, the locality of reference and replacement algorithms tailored to the workload characteristics reduce the number of physical page references that hit ratios of 95% and higher can normally be expected in the buffer. Nevertheless, large sequential scans or other "peculiar" access patterns may produce miss ratios close to 100% (in rare cases). Despite the query optimization in L5, the following extreme cases may occur for the evaluation example of Q1: All index pages and all referenced data pages are located in the DB buffer and enough memory space can be provided for the sorts. In the other extreme, each logical page reference requires a physical page reference. For large intermediate record sets, even external sort/merge operations may be anticipated.

Because modified pages have to be propagated back to external storage, the output of a modified page may precede each physical page reference to make room for the requested page in the buffer. Special L2 functionality may be dedicated to recovery provisions from failures [24]. In contrast, L1 encapsulates number, type and location of external devices. It may be implemented directly above the raw disk interface or, more frequently, perform its task in cooperation with the OS file management. Even in the L1 layer, dedicated devices, tailored file clustering or declustering measures (depending on disk access frequencies or set-oriented and parallel access interfaces [60]) or block mapping techniques regarding special workload characteristics [58] have a major impact on query processing.

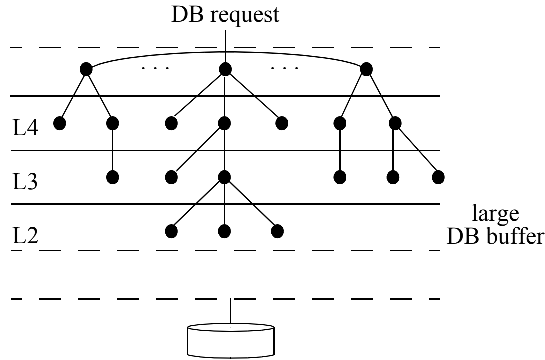

Fig. 2 Dynamic DBMS model at run time

2.3 Number of Layers Reconsidered

Ideal layers strictly enforcing the Parnas' *use* relation behave like abstract machines where the abstract machine of layer *i+1* is implemented by using the abstract machine of layer *i*. Hence, in the worst case, six formal interfaces have to be crossed before data stored on disk is reached to evaluate a DB request. This performance-critical observation prompts us to reconsider the number of system layers. On the one hand, a growing number of layers reduces the complexity of the individual layers which, in turn, facilitates system evolution. On the other hand, a growing number of interfaces to be crossed for the DB request execution increases the run-time overhead and generally reduces the DBMS optimization potential. Due to the ideal layer encapsulation, each service invocation implies parameter checking, additional data transport, and more difficult handling of non-local errors. For example, the requested data has to be copied—in its layer-specific format—from layer to layer up to the requestor. In the same way, data modified has to be propagated—again in its layer-specific format—eventually down to the disk. Hence, the number of layers seems to have a major influence on the overall system performance[4]. Above all, data copying and update propagation should be minimized across the system layers. However, our proposed architectural DBMS model is already a compromise between layer complexity/system evolution potential and request optimization/run-time overhead, as far as adequate data mapping is concerned.

Our discussion in section 2.2 already revealed the way to reduce the performance penalty introduced by the layered structure of our model. Based on these observations, we propose run-time optimizations to our static five-layer model leading to two or three effective layers in the dynamic case. As illustrated in Fig. 2, L5 is replaced by the access module whose operations directly refer to the L4 interface. At the other side, the use of a large DB buffer effectively reduces disk accesses such that almost all logical page references can be located by L2 in memory.

[4] For DBMSs, it is especially true: "Performance is not everything, but without performance everything is worth nothing."

There are precompilation approaches conceivable where the access module directly maps the SQL requests to the L3 interface to further save the crossing of the L4 interface. Hence, query preparation is pushed as far as possible. Even ad-hoc queries may be prepared in this way, because it turned out that generated access code is more effective than query interpretation. In this way, the extra cost of preparation (which extends the query response time in this case) is quickly amortized especially when large sets of records have to be accessed [7]. On the other hand, some systems pass on query preparation (at program compile time) and use—at the cost of extending the query response time—interpreters which can be understood as replacements of L5 at run time. An interpreter is a general program which, in this case, accepts any SQL statement as input and immediately produces its query resultthereby referring to the L4 interface. Mixed approaches use a preparation phase at compile time, but do not go to the extreme—the access module. They prepare intermediate query artifacts such as query graphs or execution plans in varying details, and use specific interpreters that "execute" these artifacts at run time by invoking L4 operations (and, in turn, lower-layer operations).

2.4 Binding and Information Channels

Compilation and early binding are good ideas. Of course, important aspects have to be observed when dealing with the compilation/optimization problem. Lack of run-time parameter values introduces imprecise selectivity estimates at compile time even for up-to-date statistical data. Furthermore, all preparation approaches necessarily bind their generated artifacts to the meta-data valid at preparation time, that is, compilation time in general. Later changes to the meta-data (table definition alterations, new or dropped index definitions, etc.) cannot be taken into account; hence, binding makes the artifacts data dependent and requires an automated compensation mechanism for invalidation/re-preparation. Ideally, this concept of early binding enhanced with such a compensation should combine the query evaluation efficiency of compile-time preparation with the convenience and data independence of late binding, i.e., that of a full interpreter.

So far, we have avoided to introduce control measures (dependency relationships) across the system layers, typically necessary to increase query performance/throughput or to guarantee specific system properties, e.g., dependencies to realize some functionality for load control or ACID protection. Hence, a real DBMS implementation will soften the claims of our explanation model strictly adhered to the information hiding and hierarchical layer principles. Normally, a few carefully selected dependencies or mechanisms have to be provided for control and optimization measures across system layers, that is, we need a few "vertical" information channels. For example, when the *hot set model* [52] is used to make frame reservations supporting specific QEPs in the DB buffer, query optimization in L1 has to communicate some hints to L4. On the other hand, thrashing behavior in the DB buffer or extreme situations for lock contention (occurring in lower layers) need an information channel to the load control and transaction entry queue allocated in L1.

2.5 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.

20 years ago, SQL—not standardized at that time—and the underlying relational model were simple. Today, we have to refer to SQL:1999 or SQL:2003 and an object-relational model which are complex and not well understood in all parts. Many of the new aspects and functions—such as user-defined types, type and table hierarchies, recursion, constraints, triggers—have to be adjusted in L5. While initially query translation and optimization started with solid foundations [55, 39, 34], enabled the integration of new mechanisms, and could be successfully improved [41], in particular, by using refined statistics (in particular, histograms [35]), some of the new language concepts turn out to be very hard for the optimization. For example, SQL's new landmark concept of user-defined types—despite some proposals such as enhanced abstract data types [56]—is not solved at all. Rule-based optimizers are here foredoomed because sufficiently general rules cannot be provided for them. In contrast, user-defined types have to carry their own cost model to be integrated by cost-based optimizers, which is no general proceeding: first, the "type" writers have to deliver the complex cost models and, second, many of the models' parameters are hard to determine at run time.

Work on building effective optimizers for dynamic QEPs to address the problem of changes in resource availability [17] or to "reduce the braking distance" of the query engine [6] is in progress, but far away from the point where practical techniques are ready for their use in products. Proposals typically leave resource availability to the individual algorithms, rather than to arrange for dynamic plans with alternative algorithms or alternative plan shapes. Available adaptive techniques (in L4) are not only related to the internal behavior of single algorithms (operators), but also among operators within a single query and among multiple concurrent queries. Furthermore, new dynamic optimization opportunities come up, for example, for special query types where the result set is characterized by "N tuples only" or "top/bottom N tuples". The challenge for the optimizer is to provide dynamic QEPs that try to stop the evaluation when the size N is reached. New partition-based techniques or even a kind of "gambling" help to reduce the wasted sorting and/or joining effort.

DB research delivered new algorithms to improve and extend the functionality in L4 for SQL's standard operations required. Hash joins are a success [57] and complement each sufficiently broad and efficient DBMS implementation. Furthermore, functionality for "arbitrary" join predicates, reuse of intermediate query evaluation results, sorting (internally usually optimized for relatively small sets of variable length records in memory as well as external sort/merge), etc. was improved and much better integrated. In particular, space-adaptable algorithms contribute to great improvements and support load balancing

and optimized throughput, even for high multi-programming levels [16]. For example, the replacement selection sort, which can react to presortedness, dynamically adjusts its memory requirements to save merge runs [18]. Other adaptive techniques for operators 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, etc. [17]. On the other hand, the language extensions mentioned above enforced entirely new functionality to be provided by L4. Support for some of them, close enough to the original concepts of the relational model, is successfully made available and extends the spectrum of algorithms towards enabling (originally called) non-standard applications [26]. Examples include spatial joins [5, 21] or operations supporting functionality for OLAP or data warehouses [36]. However, it seems to be inadequate or even impossible to integrate adjusted operators for the "exploding" set of new types, e.g., video, image, text, audio (referenced by the acronym VITA) and others. For the management and integration of these kinds of types, see section 5.2.

All these operations have to be "made efficient" by providing appropriate access paths and storage structures in L3. A "firestorm" of research in the last two decades tried to respond to this challenge—partly because the search problems to be solved could be described in isolation and empirical results could be produced by various kinds of simulation, that is, without the need to embed the structures into an existing DBMS. A survey article on multidimensional access methods [15] compared a large set of structures and their genealogy (with about 200 references published until 1998). Besides the ubiquitous B-tree and maybe some variants of the R-tree and the UB-tree [2], why did these structures not make their way into L3? Of course, a few of them are successfully integrated into specialized data handling system (Grid, Hashing). However, the lion's share of the proposed access paths fails to pass the DBMS acid test for various reasons. A suitable method should not be data dependent (on the value set or UID sequence) and should not require static reorganization. Moreover, it should not rely on special data preparation and its optimal use should not require the knowledge on system internals or expert experience. Furthermore, over-specialized use and tailoring to narrow applications do not promise practical success in DBMSs[5]. Finally, many of these methods (often Ph.D. proposals) disregard the DBMS environment, where dependencies to locking and recovery issues, integration into optimizer decisions, support of mixed and unexpected workload characteristics have to be regarded.

The most drastic improvements occurred at the L2 level—without much endeavor of the DB researchers. Moore's Law did the job, because the available memory is now increased by a factor of 10^4. Therefore, DB buffer sizes may now range in the area of up to 1–10 million frames while, in the same period, the individual frame size has grown from 2K to 8K–32K bytes. The classical demand-driven replacement algorithms were enhanced by ideas combining LRU together with reference density, e.g., in the form of LRU-K [45]. Furthermore, they were complemented by various prefetching algorithms and pipelined double buffering which together can automatically detect and optimize scan-based opera-

[5] Optimal support of point queries in high-dimensional spaces (say k in the range of 10–20) and nothing else is not a broad requirement.

tions. Buffer allocation algorithms, on the other hand, were proposed to exploit knowledge of declarative, set-oriented DB languages and may be used in the form of the hot set model. Finally, the huge DB buffer capacity facilitated the provision of buffer partitions where each partition can individually be tailored to the anticipated locality behavior of a specific workload. Nevertheless, buffering demands of VITA applications, considered in various projects [40], cannot be integrated in any reasonable way into L2, let alone the transfer of the huge data volumes through the layered architecture up to the application. Again, specialized handling is mandatory, as discussed in section 5.2.

While the separation of segments/pages (L2) and files/blocks (L1) opens opportunities for sophisticated data mappings and update propagations, nothing has really happened in this part of the architecture. The old and elegant concepts of *shadow pages* and *differential files*, which allow for *Atomic* update propagation and, depending on the selected checkpoint mechanism, for materialized DB states of guaranteed consistency after crashes, were considered too expensive in the normal DB processing mode. Because failures are very rare events, normal update propagation uses update-in-place (*NonAtomic*) and is performed in some optimistic way—with logging as the only failure precaution –, and more burden is shifted to the recovery and restart phases.

To conclude our pass through the DBMS layers, L1 was not a focus of interest for DB researchers. OS people proposed various improvements in file systems where only some were helpful for DB management, e.g., distribution transparency. Log-structured files [51], for example, turned out to be totally unsuitable. Furthermore, there is still no transaction support available at this layer of abstraction. However, standard file mapping was considerably refined - now supporting long fields or large objects (*Blob, Clob, DClob*) up to 2G bytes. A lot of new storage technology was invented during the last two decades—disks of varying capacity, form and geometry, DVDs, WORM storage, electronic disks, etc.. Their integration into our architectural model could be transparently performed in L1 as far as the standard file interfaces were concerned. New opportunity arrived with the disk arrays [48] supporting different clustering and declustering strategies at the file level [61, 62]. To enable parallel access, file interfaces had to be extended at the L1 interface. Their use, possibly exploiting set-oriented and parallel I/O requests, has led to new algorithms to be allocated in L2.

3 Layered Model and Transactional Properties

So far, we have discussed an explanation model for the data abstraction hierarchy in a DBMS. In particular, we have sketched the state of the art reached after 20 or more years of research and development. However, we have excluded all considerations of the failure case and the interplay of the transactional properties with the operations at the different abstraction levels. As far as ACID is concerned, the layered model again helps to describe the concepts and to derive appropriate solutions.

3.1 Atomicity

All transactional properties are not “natural” properties of computer systems. In the presence of failures we even cannot guarantee atomic execution of instructions or I/O operations. Hence, these properties have to be accomplished by complex software mappings. Early attempts to provide atomicity were based on the recovery blocks [50]. However, they were impractical already for atomic actions in a single layer, let alone for DML operations executed across several layers or even a set of separate DML operations bracketed into a transaction. Although atomic propagation of modified blocks/pages can be met by particular update propagation schemes [24, 19], transaction atomicity must be achieved through a two-phase commit (2PC) protocol by combined use of concurrency control and logging/recovery measures.

3.2 Consistency

The *C* in ACID guarantees DB schema consistency which is preserved by every successful transaction. To develop a framework for transaction implementation, we will refine our notion of consistency. For this purpose, it is helpful to introduce a hierarchy of operations—I/O operations, elementary actions, actions, DML operations, transactions—which corresponds to our layered model, a simplified version of which is sufficient for the refined consideration of transactional aspects. An obvious, but nevertheless important observation is that a data granule to which an operation has to be applied must be consistent w.r.t. this operation (operation consistency, layer-specific consistency). After successful execution it is again operation consistent. In this sense, a transaction can be explained as a hierarchy of nested atomic actions. Hence, object consistency needs layer-specific consistency of all layers below. At each level in Fig. 3, we give an example for an operation that requires the given type of consistency and which preserves it after successful (atomic) completion.

To reduce the complexity of discussion, Fig. 3 simplifies our five-layer model to the well-known three-layer model and we will refer to it when appropriate. The lowest layer, called *storage system*, comprises L1 and L2. For our ACID considerations, the separation of blocks and pages would not capture new aspects. On the other hand, in L3, called *access system*, we distinguish between elementary action consistency (EAC) and action consistency. A single action (at L3) may cause updates of several pages, for example, when a B-tree insertion causes page splits, whereas an elementary action is always confined to a single page.

Again, a separate consideration of L4 and L5 would not reveal new insights for ACID, because their operations affect multiple pages in general. L4 roughly corresponds to a navigational one-record-at-a-time DBMS interface (e.g., for the network or (simple) object-oriented data models), whereas L5 characterizes declarative set-oriented DBMS interfaces (e.g., SQL). As indicated by the upper layer in Fig. 3, called *data system*, single DML operations require and guarantee (DBMS) API consistency. This relationship is emphasized by the golden rule[6] of C. J. Date [10] and explains why integrity constraints attached to atomic DML operations (statement atomicity in SQL) have to be satisfied at end of oper-

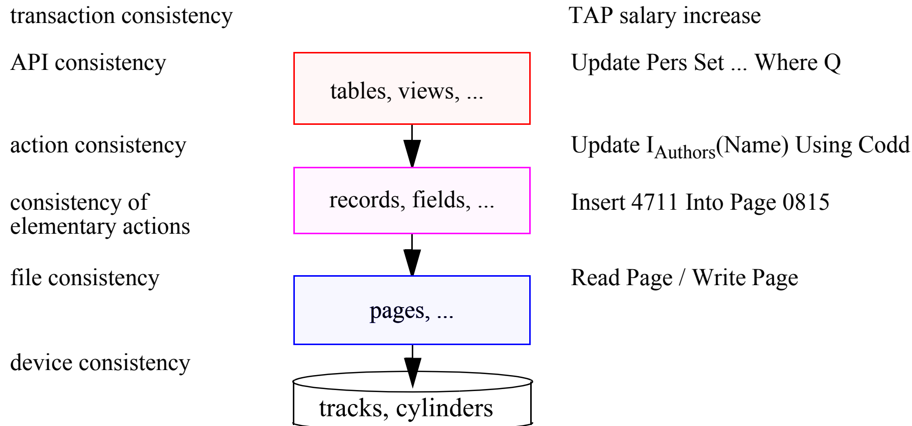

Fig. 3 System layers and consistency

ation (e.g., a complex case of it are referential actions). In turn, only "higher" DB schema constraints can be declared *deferrable*, to be satisfied later and checked at the latest in the commit phase to fully guarantee transaction consistency.

3.3 Isolated execution

By passing the *I* in ACID, we only remark that the operation hierarchy in Fig. 3 is appropriate to explain the various conflict serializability models in the context of a DBMS. The schedules/histories of the page model could be derived by a history writer observing the interface between access system and storage system, whereas other models would choose the interfaces of the operations considered. Because only the specification of conflict relations among concurrent operations—but not their specific semantics—is needed for conflict serializability, appropriate protocols achieving transaction isolation can be provided at any abstraction level. For a comprehensive discussion see the textbook of Gerhard Weikum and Gottfried Vossen [64].

20 years ago, multi-granularity locking was the method of choice for multi-user synchronization in DBMSs and, surprisingly, it still is—extended by a larger set of specialized lock modes. While early DBMSs used the page granule as the smallest lockable unit, today records or even smaller units are a must to cope with resource contention in an acceptable manner [44]. Of course, practical progress has been made on efficiently synchronizing operations on special structures (indexes, trees, hot spots) and for specific sequences of access requests [42, 43]. Furthermore, multi-version methods currently seem to gain more

---

[6] "No update operation must ever be allowed to leave any relation or view in a state that violates its own predicate. Likewise no update transaction must ever be allowed to leave the database in a state that violates its own predicate."

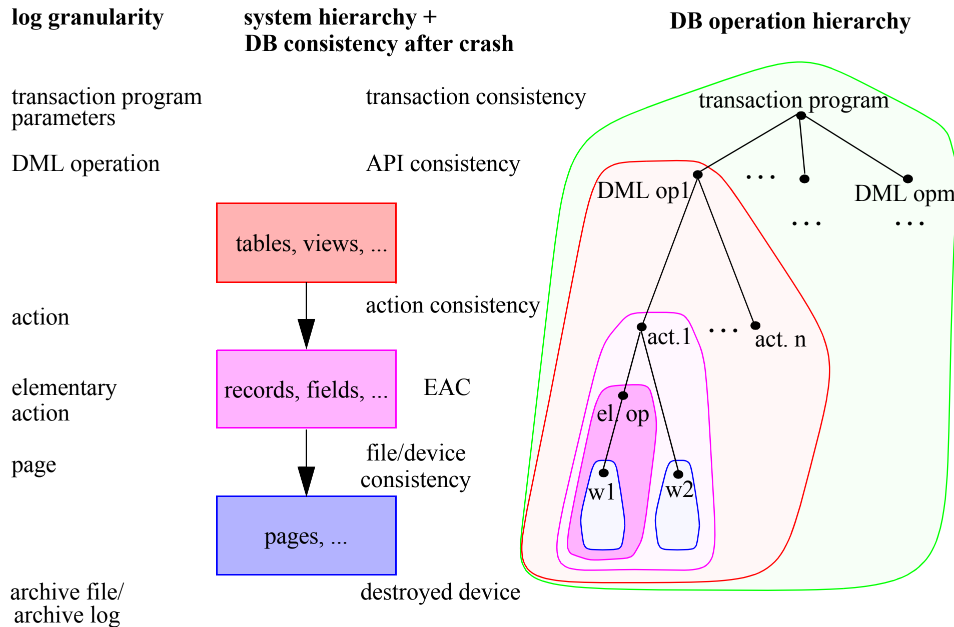

Fig. 4 DB consistency at restart and logging

importance, because plenty of memory allows keeping multiple versions per object to increase the effective level of transaction parallelism. However, considering the myriads of conflict-serializable synchronization protocols proposed [59], it is humbling how few of these ideas have entered the DBMS world.

3.4 Durability

Recording of redundancy during normal processing—to be prepared for the event of failures—and an application of recovery algorithms in case of a failure remain the standard measures to achieve durability. Recovery from failures always aims at the most recent transaction-consistent state of the DB. Because the level-specific operations preserve layer-specific consistency for their data, they can be exploited to extract logging information. Hence, logging can be performed at each level, as illustrated in Fig. 4.

3.4.1 Logging

Log data is collected during normal processing and applied by the recovery algorithms in case of a failure. While transaction recovery can refer to the current state of the DB and use context information, recovery from a crash or a media failure must use the materialized DB or the archive DB, respectively. Hence, in these cases the DB has to satisfy the corre-

sponding layer-specific consistency required for the successful application of the log information to redo or undo the transaction effects at the chosen abstraction level. Therefore, we establish a strong and performance-critical relationship, in particular, for the crash recovery if we select a specific logging method. Furthermore, logging has to observe a number of rules such that transaction-oriented recovery is possible at any rate [24].

Physical logging is sufficient when device or file consistency can be expected at the time of crash recovery (restart). In turn, EAC enables *physiological logging* and LSNs [19] which leads to the practically most important logging/recovery method. Because the effects of an elementary action are always confined to a page, non-atomic propagation of pages to disk is sufficient to enable the corresponding Redo and Undo operations in history sequence. Hence, physiological logging[7]—physical to a page, logical within a page thereby tolerating displacements and rearrangements of data within a page—can be applied. Furthermore, the use of log sequence numbers (LSNs) allows a simple check at restart of whether or not the modifications of an elementary action have reached the materialized DB. In contrast, *logical logging* implies the atomic propagation of all pages modified by the corresponding operation to successfully perform Redo or Undo recovery for such operations (actions and operations at higher levels).

The question is which kind of consistency for the materialized DB can be guaranteed at restart, i.e., after a crash has occurred? Table 2 summarizes the required logging methods for a given consistency level.

Table 2 Consistency and logging

consistency level at restart	adjusted log information
file consistency	pages (before- and after-images)
elementary action consistency	physiological logging
action consistency	actions (entries)
API consistency	DML operations
transaction consistency	transaction program invocations with params

The converse conclusion is, however, not compelling. For example, if we use DML operation logging, we do not automatically assure the API consistency for the materialized DB. Therefore, extra precautions are needed during normal processing. Usually, the higher the consistency level guaranteed, the more expensive are the mapping and propagation algorithms (e.g., shadow pages, checkpoints) to establish the respective level. If we can rely on a consistency level at restart, it is possible to choose logging methods corresponding to a lower consistency level. However, this idea is not cost-effective, because logging costs typically decrease with increasing consistency levels.

[7] Its implementation could be shifted to the buffer manager (fix page, ..., write log), because only objects in individual pages are involved [29].

How do we establish a consistency level of the materialized DB? If a device is destroyed, we cannot provide the minimum consistency for the logging methods discussed. Therefore, we must usually[8] perform recovery algorithms tailored to device failures, the so-called media recovery with archive DB and archive log. In the following, we assume file consistency as a minimum consistency level.

3.4.2 Non-atomic Update Propagation

If we use *non-atomic propagation* methods, the DB is only file consistent when a crash occurs; all individual blocks are readable, but they are nicknamed as *chaos consistent*, because the DB contains actual, outdated, and invalid blocks. But the correctness of non-atomic propagation methods does not rely on specific requirements of page I/O, because the corresponding recovery methods based on logging of pages or elementary actions can be correctly applied to individual pages only.

If all before- and after-images of the modified pages are logged, which is extremely expensive and log-space consuming, entire pages can be exchanged and transaction-oriented recovery is possible. Another significant cost factor is given by the minimum lock granule[9] implied by page locking. Physiological logging brings a substantial improvement, because an Undo and a Redo of modifications in elementary-action-consistent pages can be performed based on a space-saving logging method. Using LSNs, the recovery manager can efficiently decide whether an Undo or a Redo operation has to be applied to a page addressed, even for lock granules smaller than a page.

3.4.3 Atomic Update Propagation

Because operations in higher system layers may affect multiple pages, the corresponding recovery operations based on the resp. logging methods imply the existence of the entire data granule. For this reason, the set of resp. pages must be completely or not at all in the materialized DB, a property which can only be obtained by checkpointing and atomic propagation methods [24]. Referring to Fig. 4, action consistency for pages to be recovered is accomplished when the set of pages involved in an update action is either completely in the materialized DB or not at all; this is obtained by action-consistent checkpoints. Then all effects of the actions starting from this checkpoint can be repeated or undone on this DB state. The same is true for DML operations with API-consistent checkpoints or transactions with transaction-consistent checkpoints. Because only entire pages can be written to disk, checkpointing has to be synchronized with concurrency control; otherwise, large lock granules (at least page locks) have to be applied which enforce serial operations in pages.

It is interesting to note that higher-level operation-consistent DB states can be reconstructed (Redo) based on physiological logging and LSNs. As a consequence, Undo operations

[8] In special cases of destroyed blocks, page logging may work if the entire block can be replaced.
[9] The lock granule must be larger or equal to the log granule used [24], see appendix.

are possible using operation-consistent logical logging methods [29]. As a prerequisite, only pages with operation-complete modifications must reach the disk which can be achieved by fixing all pages involved in an update until the end of the specific operation (which is trivially satisfied for elementary actions). However, this idea becomes quickly impractical with growing operation granules. At the level of DML operations, this would, for example, require a "long-term " buffer fixing of all pages involved in a set-oriented update request.

3.4.4 The Winner Solution

In early DBMSs, logging/recovery schemes were rather simple, exhibiting little or no optimization. They were often dominated by non-atomic propagation and page logging which implied page locking as a minimal granularity (see appendix). Therefore, they were neither efficient nor elegant. An exception was System R which implemented an atomic recovery scheme [4] whose atomicity mechanism was based on shadow pages [38]. Empirical studies [8] certified its high overhead during normal processing. Furthermore, the presence of rapidly growing DB buffers made direct checkpoints infeasible for interactive DB processing. As a consequence, no effective solution exists for atomic propagation schemes eliminating, in practice, all recovery methods which require higher DB consistency levels at restart (logging granularity: transaction, DML operation, action). It is safe to say that *all clean and elegant crash recovery solutions do not pay off* (e.g., *Atomic, NoSteal, Force* [24]). Hence, they disappeared from the DBMS world. Nowadays, non-atomic propagation, also compatible with indirect checkpointing techniques, is the clear winner in this area. Hence, the best performing recovery schemes, which effectively cope with huge buffers and unburden normal processing with logging-related I/O overhead, are characterized by *NonAtomic, Steal, NoForce* supported by Fuzzy checkpoints and some more I/O saving tricks [43].

4 Architectural Variants

Up to now, we have intensively discussed the questions of data mapping and transactional support in a centralized DBMS architecture. In the last two decades, however, a variety of new data management scenarios emerged in the DBMS area. How can these be linked to the core issues of our architectural discussion so far?

A key observation is that the *invariants in database management determine the mapping steps of the supporting architecture*. In our case, we started with the primary requirement of navigational or set-oriented processing of record-like data which led to the layered architecture sketched in Fig. 2 and 4. In many of the new data management scenarios, the basic invariants still hold true: page-oriented mapping to external storage, management of record-oriented data, set-oriented database processing. Hence, we should be able to identify the resp. layers/components in the evolved architectures and to explain the similarity in database processing using their architectural models.

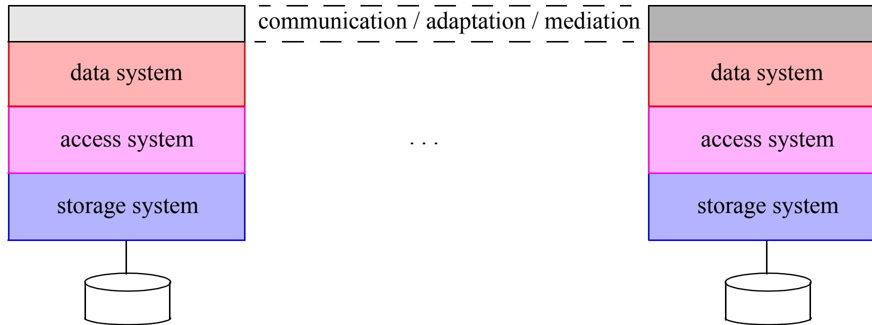

Fig. 5 Horizontal DBMS distribution

4.1 Horizontal Distribution of DB 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 an 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.

Shared-nothing DBMSs partition their data and need some functionality to decompose DB requests, forward them to the corresponding node, and assemble the answers to the query result thereby providing a local and homogeneous view of the entire DBMS to the user (single system view). While the functionality of the individual nodes essentially remains unchanged, some new cross-node tasks are needed for optimal DBMS processing, e.g., load balancing, global query optimization in addition to the local one, deadlock detection, 2PC protocol, global recovery precautions, etc. [37]. For *shared-disk DBMSs*, the adjustment and coordination aspects are primarily in the area of buffer management and once more in the failure recovery from individual node crashes [49]. In contrast, *parallel DBMSs* provide their services to run identical operations on partitioned data in parallel (data parallelism) or to apply intra-operation parallelism to the same data. Hence, the major challenge is to decompose a problem for a large number of processors and to coordinate them such that a linear scale-up or speed-up is achieved [12].

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 DBMSs* represent the entire spectrum of possible data integration scenarios and usually need an adjustment of the DB requests at the level of the data model [49] or a compensation of functionality not generally available. As opposed to the distributed homogeneous DBMSs, 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 amang the federation scenarios is represented by

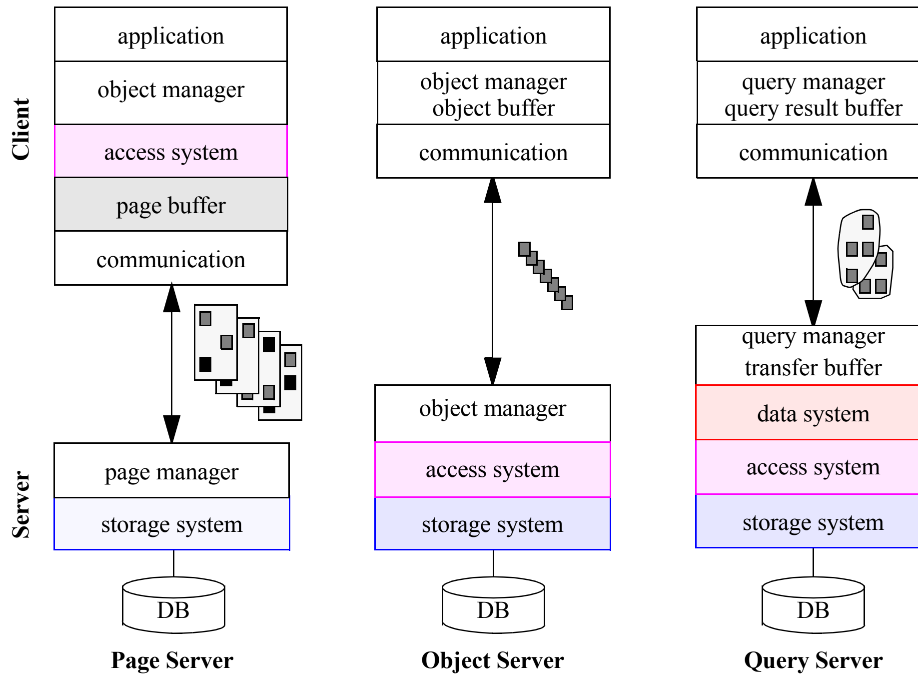

Fig. 6 Architectural variants of client/server DBMSs

Multi-DBMSs, for which the connection layer primarily takes over the role of a global transaction manager passing unmodified DB requests to the participating DB servers [54].

4.2 Vertical Distribution of DBMS Processing

The typical (most important) representatives of this class of DBMS architectures belong to the 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. Hence, the underlying data management scenarios are tailored to engineering applications [23]. As indicated in Fig. 6, various forms of this architectural variant exist [28]. They are characterized by DBMS-controlled data or object buffers at the client side to exploit data reference locality as the major mechanism to enhance performance. The most sophisticated one is the query server, in its functionality comparable to DBMS kernel architectures [53]. Its real challenge is declarative, set-oriented query processing thereby using the current content of the query result buffer [11].

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 DB frontends

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, for example, span several tables organized as cache groups by equi-joins [31]. While the locality preservation of the query result buffer in query servers can take advantage of application hints [11], *adaptivity* of database caching is a major challenge for future research [1]. 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 [22].

5 New Architectural Requirements

So far, our architectural layers 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 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.

5.1 XTC Architecture

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 [14].

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 (XDBMSs) emerged in recent years, an architectural example of which is illustrated in Fig. 7. The current state of the XTC architecture (XML Transaction Controller [30]) perfectly proves that native XDBMSs can be implemented along the lines of our five-layer architecture. Special mappings of XML documents to trees of records and, in turn, to container files which contribute to excellent storage utilization [30] as well as fine-grained concurrency control are characteristic for the XTC architecture. Furthermore, the use of the Dewey numbering system supports dynamic modifications of the XML structure at any document location thereby enabling the setting of locks along the entire ancestor path (through their identifier structure) without the need to access the XML document [29].

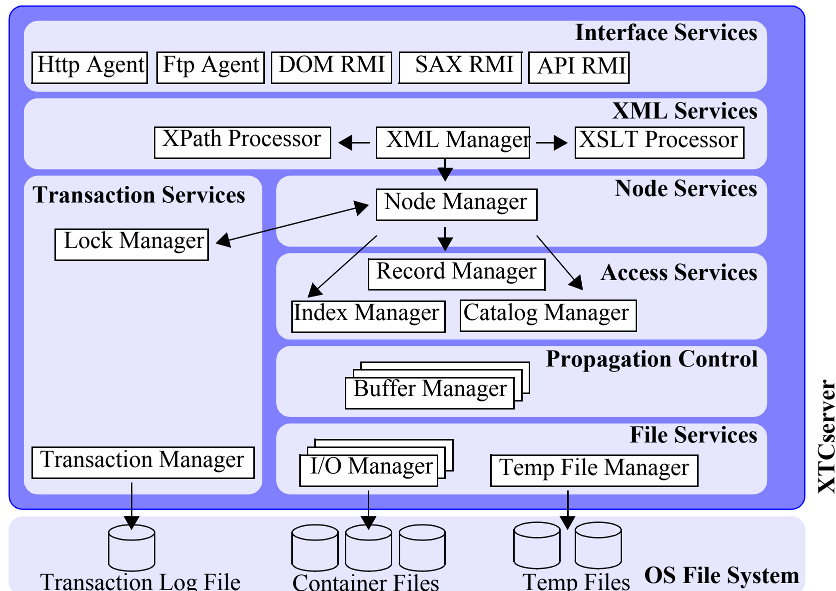

Fig. 7 XTC system—overview

The issues of a simultaneous support of XML and relational database management were explored in [32]. 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. 7.

5.2 The Next Database Revolution Ahead?

The discussion above has revealed that *even XML data* cannot be adequately integrated into the original layer model because the processing invariants, especially those of access and data system, valid in record-oriented DBMS architectures do not hold true for document trees with other types of DB requests. Furthermore, what has to be done when the conceptual differences of the data types such as VITA or data streams are even larger?

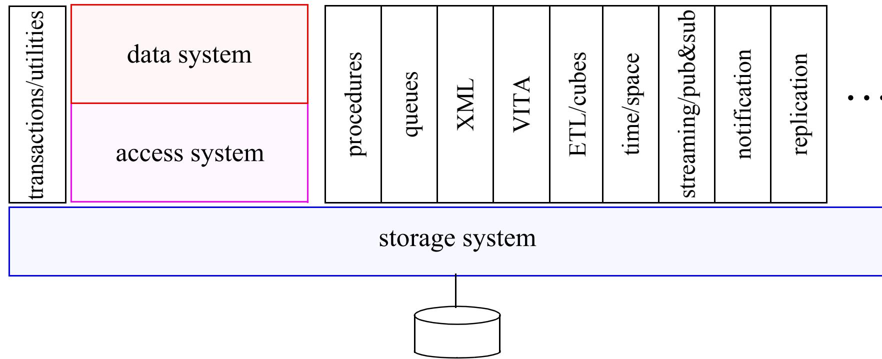

Fig. 8 Desirable extensions for future DBMS architectures

Because the new data types can often only reuse 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) 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. 8 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 [20]. 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" [20]. Furthermore, because they cooperate on behalf of applications, ACID protection has to be assured for all messages and data taking part in a transaction.

5.3 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 systems. Typically, adaptable component (layer) 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. For example, optimized dynamic index selection, i.e., whether a new index for a table is cost-effective, cannot be performed by the index manager alone. Several components have to collect statistical data to 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. For these reasons, online feedback control loops consisting of observation, analysis, planning, and reaction were proposed for achieving the "self-*" system properties in [63] which, however, amplify the information channels across system layers.

On the other hand, 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 d*ependability* objective which is much broader than self-tuning or self-administration. To develop future DBMSs which can be defined as "dependable = self-* + trouble-free" , Gerhard Weikum et al. call for a highly componentized system architecture with small, well-controlled component interfaces (as narrow as possible) and limited and relatively simple functionality per component which implies the reduction of optional choices [63]. On the other hand, most self-* properties are not easily amenable to mathematical modeling and run-time analysis, because they are non-functional in general. 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.

6 Conclusions and Outlook

In the tradition of [3], we looked at the progress of database research and development which happened in the 20 years of history of the BTW—the German conference "Database Systems in Business, Technology, and the Web". Because of the growing breadth and depth of the database area, this paper primarily focused on the developments concerning the DBMS architecture for the declarative and set-oriented (relational) processing paradigm of record-like structures. We showed that the five-layer hierarchical model proposed more than 20 years ago was able to accommodate all the extensions and optimizations for the originally established paradigm. Even new data management scenarios incorporating the processing invariants of our five-layer model could be well embraced by architectural variants of it.

The more the integration of new data types required new processing invariants, the less our architecture was able to accommodate them. It is an open question how the architecture for an e*xtensible object-relational system where non-procedural relational operators manip-*

ulate 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.

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

Appendix: The Ten Commandments[10]

General Rules

- I. Recovery based on logical logging relies on a matching operation-consistent state of the materialized DB at the time of recovery.
- II. The lock granule must be at least as large as the log granule.
- III. Crash recovery under non-atomic propagation schemes requires Redo Winners resp. Redo All (repeatable history) before Undo Losers, whereas the order of Undo and Redo is irrelevant under atomic schemes.

Rules for Undo Recovery

- IV. State logging requires a WAL protocol (if pages are propagated before Commit).
- V. Non-atomic propagation combined with logical logging is generally not applicable (for Redo and Undo recovery).
- VI. If the log granularity is smaller than the transfer unit of the system (block size), a system crash may cause media recovery.
- VII. Partial rollback within a transaction potentially violates the 2PL protocol (programming discipline necessary).

Rules for Redo Recovery

- VIII. Log information for Redo must be collected independently of measures for Undo.
- IX. Log information for Redo must be written at the latest in phase 1 of Commit.
- X. To guarantee repeatability of results of all transactions using Redo recovery based on logical logging, their DB updates must be reproduced on a transaction basis (in single-user mode) in the original Commit sequence.

---

[10] Theo Härder, Andreas Reuter: A Systematic Framework for the Description of Transaction-Oriented Logging and Recovery Schemes, Internal Report DVI 79-4, FG Datenverwaltungssysteme I, TH Darmstadt, Dec. 1979. Commandments I and V are valid for logical and physical transition logging. The latter based on EXOR differences does not seem to be used anymore in DBMSs [28].

References

[1] Mehmet Altinel, Christof Bornhövd, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, Berthold Reinwald: Cache Tables: Paving the Way for an Adaptive Database Cache. VLDB 2003: 718-729

[2] Rudolf Bayer: The Universal B-Tree for Multidimensional Indexing: General Concepts. WWCA 1997: 198-209

[3] Albrecht Blaser: Die BTW im Wandel der Datenbank-Zeiten. BTW 1995: 48-50

[4] Mike W. Blasgen, Morton M. Astrahan, Donald D. Chamberlin, Jim Gray, et al.: System R: An Architectural Overview. IBM Systems Journal 20(1): 41-62 (1981)

[5] Thomas Brinkhoff, Hans-Peter Kriegel, Bernhard Seeger: Efficient Processing of Spatial Joins Using R-Trees. SIGMOD Conference 1993: 237-246

[6] Michael J. Carey, Donald Kossmann: Reducing the Braking Distance of an SQL Query Engine. VLDB 1998: 158-169

[7] Donald D. Chamberlin, Morton M. Astrahan, W. Frank King III, Raymond A. Lorie, James W. Mehl, Thomas G. Price, Mario Schkolnick, Patricia G. Selinger, Donald R. Slutz, Bradford W. Wade, Robert A. Yost: Support for Repetitive Transactions and Ad Hoc Queries in System R. ACM Trans. Database Syst. 6(1): 70-94 (1981)

[8] Donald D. Chamberlin, Morton M. Astrahan, Mike W. Blasgen, Jim Gray, et al.: A History and Evaluation of System R. Commun. ACM 24(10): 632-646 (1981)

[9] Douglas Comer: The Ubiquitous B-Tree. ACM Comput. Surv. 11(2): 121-137 (1979)

[10] Chris J. Date: An Introduction to Database Systems, 6th Edition. Addison-Wesley 1995

[11] Stefan Deßloch, Theo Härder, Nelson Mendonca Mattos, Bernhard Mitschang, Joachim Thomas: Advanced Data Processing in KRISYS: Modeling Concepts, Implementation Techniques, and Client/Server Issues. VLDB J. 7(2): 79-95 (1998)

[12] David J. DeWitt, Jim Gray: Parallel Database Systems: The Future of High Performance Database Systems. Commun. ACM 35(6): 85-98 (1992)

[13] Edsger W. Dijkstra: Letters to the editor: The Goto Statement Considered Harmful. Commun. ACM 11(3): 147-148 (1968)

[14] Daniela Florescu, Donald Kossmann: Storing and Querying XML Data using an RDMBS. IEEE Data Eng. Bull. 22(3): 27-34 (1999)

[15] Volker Gaede, Oliver Günther: Multidimensional Access Methods. ACM Comput. Surv. 30(2): 170-231 (1998)

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

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

[18] Goetz Graefe: Implementation of Sorting in Database Systems, submitted 2003.

[19] Jim Gray, Andreas Reuter: Transaction Processing: Concepts and Techniques. Morgan Kaufmann 1993

[20] Jim Gray: The Next Database Revolution. SIGMOD Conference 2004: 1-4

[21] Oliver Günther: Efficient Computation of Spatial Joins. ICDE 1993: 50-59

[22] Hongfei Guo, Per-Ake Larson, Raghu Ramakrishnan, Jonathan Goldstein: Relaxed Currency and Consistency: How to Say "Good Enough" in SQL. SIGMOD Conference 2004: 815-826

[23] Theo Härder, Christoph Hübel, Klaus Meyer-Wegener, Bernhard Mitschang: Processing and Transaction Concepts for Cooperation of Engineering Workstations and a Database Server. Data Knowl. Eng. 3: 87-107 (1988)

[24] Theo Härder, Andreas Reuter: Principles of Transaction-Oriented Database Recovery. ACM Comput. Surv. 15(4): 287-317 (1983)

[25] Theo Härder, Andreas Reuter. Concepts for Implementing a Centralized Database Management System. Proc. Int. Computing Symposium on Application Systems Development, March 1983, Nürnberg, B.G. Teubner-Verlag, 28-60

[26] Theo Härder, Andreas Reuter: Database Systems for Non-Standard Applications, in: Proc. Int. Computing Symposium on Application Systems Development, March 1983, Nürnberg, B.G. Teubner-Verlag, 452-466

[27] Theo Härder, Andreas Reuter: Architektur von Datenbanksystemen für Non-Standard-Anwendungen. BTW 1985: 253-286

[28] Theo Härder, Erhard Rahm: Datenbanksysteme: Konzepte und Techniken der Implementierung, 2. Auflage, Springer 2001

[29] Michael Haustein: Eine XML-Programmierschnittstelle zur transaktionsgeschützten Kombination von DOM, SAX und XQuery, in: Proc. BTW, Karlsruhe, March 2005

[30] Micheal Haustein, Theo Härder: Fine-Grained Management of Natively Stored XML Documents, submitted (2004), http://wwwdvs.informatik.uni-kl.de/pubs/p2004.html

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

[32] Alan Halverson, Vanja Josifovski, Guy Lohman, Hamid Pirahesh, Mathias Mörschel: ROX: Relational Over XML. Proc. 30th Int. Conf. on Very Large Data Bases (VLDB'04), Toronto (2004)

[33] IBM DB2 Universal Database (V 8.1). http://www.ibm.com/software/data/db2/

[34] Matthias Jarke, Jürgen Koch: Query Optimization in Database Systems. ACM Comput. Surv. 16(2): 111-152 (1984)

[35] Yannis E. Ioannidis: The History of Histograms (abridged). VLDB 2003: 19-30

[36] Wolfgang Lehner: Datenbanktechnologie für Data-Warehouse-Systeme. dpunkt 2002

[37] Bruce G. Lindsay: A Retrospective of R*: A Distributed Database Management System. Proceedings of the IEEE 75(5): 668-673 (1987)

[38] Raymond A. Lorie: Physical Integrity in a Large Segmented Database. ACM Trans. Database Syst. 2(1): 91-104 (1977)

[39] Raymond A. Lorie, Bradford W. Wade: The Compilation of a High Level Data Language. IBM Research Report RJ2598: (1979)

[40] Ulrich Marder, Detlef Merten: Systempufferverwaltung in Multimedia-Datenbanksystemen. BTW 1995: 179-193

[41] Bernhard Mitschang: Anfrageverarbeitung in Datenbanksystemen - Entwurfs- und Implementierungskonzepte. Vieweg 1995

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

[43] C. Mohan, Donald J. Haderle, Bruce G. Lindsay, Hamid Pirahesh, Peter M. Schwarz: ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. ACM Trans. Database Syst. 17(1): 94-162 (1992)

[44] C. Mohan: Repeating History Beyond ARIES. VLDB 1999: 1-17

[45] Elizabeth J. O'Neil, Patrick E. O'Neil, Gerhard Weikum: The LRU-K Page Replacement Algorithm For Database Disk Buffering. SIGMOD Conference 1993: 297-306

[46] David L. Parnas: On the Criteria To Be Used in Decomposing Systems into Modules. Commun. ACM 15(12): 1053-1058 (1972)

[47] David L. Parnas, Daniel P. Siewiorek: Use of the Concept of Transparency in the Design of Hierarchically Structured Systems. Commun. ACM 18(7): 401-408 (1975)

[48] David A. Patterson, Garth A. Gibson, Randy H. Katz: A Case for Redundant Arrays of Inexpensive Disks (RAID). SIGMOD Conference 1988: 109-116

[49] Erhard Rahm: Mehrrechner-Datenbanksysteme - Grundlagen der verteilten und parallelen Datenbankverarbeitung. Addison-Wesley 1994

[50] Brian Randell, P. A. Lee, Philip C. Treleaven: Reliability Issues in Computing System Design. ACM Comput. Surv. 10(2): 123-165 (1978)

[51] Mendel Rosenblum, John K. Ousterhout: The Design and Implementation of a Log-Structured File System. ACM Trans. Comput. Syst. 10(1): 26-52 (1992)

[52] Giovanni Maria Sacco, Mario Schkolnick: A Mechanism for Managing the Buffer Pool in a Relational Database System Using the Hot Set Model. VLDB 1982: 257-262

[53] Hans-Jörg Schek, H.-Bernhard Paul, Marc H. Scholl, Gerhard Weikum: The DASDBS Project: Objectives, Experiences, and Future Prospects. IEEE Trans. Knowl. Data Eng. 2(1): 25-43 (1990)

[54] Hans-Jörg Schek, Gerhard Weikum: Erweiterbarkeit, Kooperation, Föderation von Datenbanksystemen. BTW 1991: 38-71

[55] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, Thomas G. Price: Access Path Selection in a Relational Database Management System. SIGMOD Conference 1979: 23-34

[56] Praveen Seshadri: Enhanced Abstract Data Types in Object-Relational Databases. VLDB J. 7(3): 130-140 (1998)

[57] Leonard D. Shapiro: Join Processing in Database Systems with Large Main Memories. ACM Trans. Database Syst. 11(3): 239-264 (1986)

[58] Sven Tafvelin: Sequential Files on Cycling Storage. IFIP Congress 1974: 983-987

[59] Alexander Thomasian: Concurrency Control: Methods, Performance, and Analysis. ACM Comput. Surv. 30(1): 70-119 (1998)

[60] Gerhard Weikum, Bernd Neumann, H.-Bernhard Paul: Konzeption und Realisierung einer mengenorientierten Seitenschnittstelle zum effizienten Zugriff auf Komplexe Objekte. BTW 1987: 212-230

[61] Gerhard Weikum, Peter Zabback: I/O-Parallelität und Fehlertoleranz in Disk-Arrays, Teil 1: I/O-Parallelität. Informatik Spektrum 16(3): 133-142 (1993)

[62] Gerhard Weikum, Peter Zabback: I/O-Parallelität und Fehlertoleranz in Disk-Arrays, Teil 2: Fehlertoleranz. Informatik Spektrum 16(4): 206-214 (1993)

[63] Gerhard Weikum, Axel Mönkeberg, Christof Hasse, Peter Zabback: Self-tuning Database Technology and Information Services: from Wishful Thinking to Viable Engineering. VLDB 2002: 20-31

[64] Gerhard Weikum, Gottfried Vossen: Transactional Information Systems—Theory, Algorithms, and the Practice of Concurrency Control and Recovery, Morgan Kaufmann 2002[11]

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