Information System Architectures: From Art to Science

Peter C. Lockemann
Fakultät für Informatik
Universität Karlsruhe
Postfach 6980
76128 Karlsruhe
lockeman@ipd.uka.de

Abstract: The presentation claims that architectural design plays a crucial role in system development as a first step in a process that turns a requirements specification into a working software and hardware system. As such, architectural design should follow a rigorous methodology – a science – rather than intuition – an art. Our basic premise is that requirements in information systems follow a service philosophy, where services are characterized by their functionality and quality-of-service parameters. We develop a design hypothesis that takes the service characteristics into account in a stepwise fashion. We then validate the hypothesis for traditional database characteristics, demonstrate for novel requirements how these would affect architectures, and finally apply it to the current 4-tier server architectures.

1 Motivation

Information systems grow in the diversity of their application domains, number of users, and geographic distribution, but so does their complexity in terms of the number and functionality of components and the number of connections between these. An almost bewildering multitude of architectural patterns has appeared over the more recent past, that try to bring order into the evolving chaos. To name just a few of the buzzwords, take layered architectures, n-tier architectures, component architectures, middleware, vertical architectures, horizontal architectures, enterprise this-and-that. Nonetheless, it seems that these architectures have enough in common so that one suspects that they just look at similar phenomena from different perspectives, emphasize different aspects, or explore issues to different depths.

The premise of this paper is that architectural design plays a crucial role in system development. Unfortunately though, architectural system design does not seem to have too many friends. Typical excuses are that "top-down designs never work anyway because they ignore the technical possibilities and opportunities", that "even the cleanest architecture deteriorates over time due to the many additions and modifications on short notice", or that "architectures emphasize order over performance". We suspect that the real reason is the lack of a comprehensive, systematic and unifying approach to architectural design that makes the patterns in some sense comparable.

We claim that architectural design is the first step in a process that turns a requirements specification into a working software and hardware system and, hence,

could be seen as “programming-in-the-very-large”. Since it is an accepted doctrine that mistakes when caught in the early stages are much cheaper to correct than when discovered in the late stages, good architectural system design could be of enormous economical potential.

The purpose of this paper is to take a first step in the direction of a methodology for architectural design. Or in other words, we submit that architectural design should follow a methodology and not intuition, i.e., should be treated as a science and not as an art. In order not to become overly ambitious, and to stay within the confines of a conference paper, we will limit ourselves to information systems as the synthesis of data base and data communication systems, with more emphasis on the former.

2 Services

2.1 Services and resources

Since we claim that architectural design is the first step in a process that turns a requirements specification into a working software and hardware system, an essential ingredient of the design method is a uniform and rigorous requirements specification. Requirements is something imposed by an outside world. For information systems the outside world are the business processes in some real-world organization such as industry, government, education, financial institutions, for which they provide the informational support. Figure 1 illustrates the basic idea.

The counterpart of business processes in an information system are informational processes. Business processes proceed in a linear (as in Figure 1) or non-linear order of steps, and so do the informational processes. To meet its obligations, each step draws on a number of resources. *Resources* are infrastructural means that are not tied to any particular process or business but support a broad spectrum of these and can be shared, perhaps concurrently, by a large number of processes. In an information system the resources are informational in nature. Because of their central role, resources must be managed properly to achieve the desired system goals of economy, scale, capacity and timeliness. Therefore, access to each resource is through a *resource manager*. In the remainder we use the term *information systems* in the narrower sense of a collection of informational resources and their managers.

What qualifies as a resource depends on the scope of a process. For example, in decision processes the resources may be computational such as statistical packages, data warehouses or data mining algorithms. These may in turn draw on more generic resources such as database systems and data communication systems.

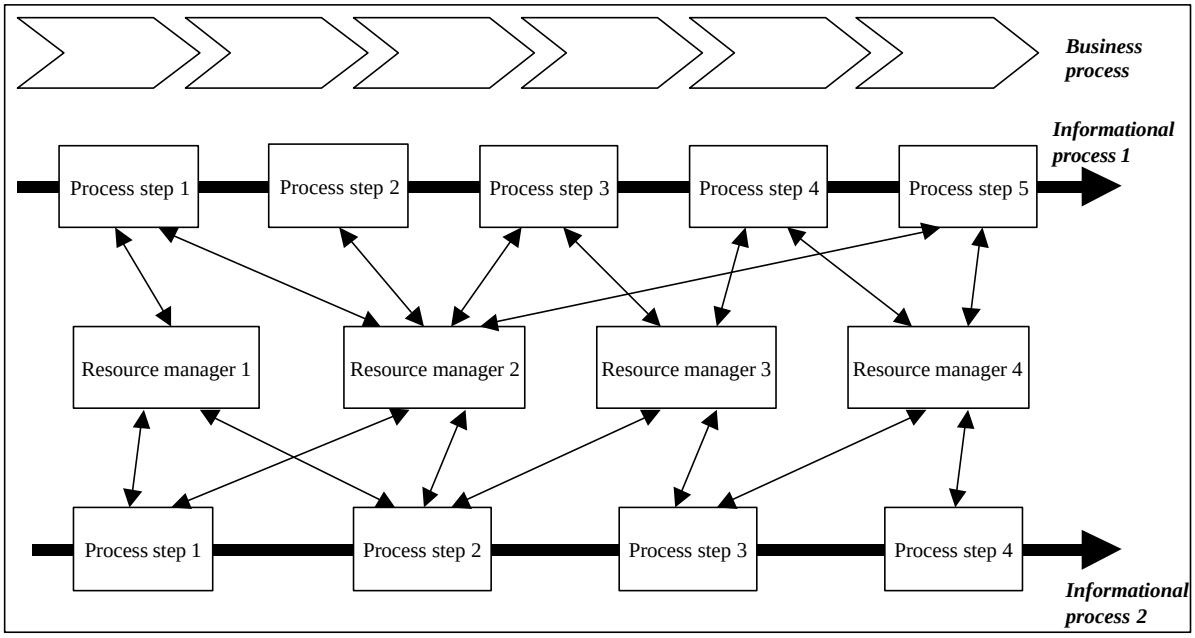

Figure 1
Business processes, informational processes and resources

What is of interest from an outside perspective is the kind of support a resource may provide. Abstractly speaking, a resource may be characterized by its *competence*. Competence manifests itself as the range of tasks that the resource manager is capable of performing. The range of tasks is referred to as a *service*. In this view, a resource manager is referred to as a *service provider* (or *server* for short) and each subsystem that makes use of a resource manager as a *service client* (or *client* for short).

2.2 Service characteristics

The relationship between a client and a server is governed by a service level agreement. In this agreement the server gives certain guarantees concerning the characteristics of the services it provides. From the viewpoint of the client the server has to meet certain *obligations* or *responsibilities*.

The responsibilities can be broadly classified into two categories. The first category is *service functionality* and covers the collection of functions available to a client and given by their syntactical interfaces (signatures) and their semantic effects. The semantic effects often reflect the interrelationships between the functions due to a shared state. Functionality is what a client basically is interested in.

The second category covers the *qualities of service*. These are non-functional properties that are nonetheless considered essential for the usefulness of a server to a client.

2.3 Service qualities

To make the discussion more targeted, we study what technical qualities of service we come to expect from an information system.

Ubiquity. In general, an information system includes a large – in the Internet even unbounded - number of service providers. Access to services should be unrestricted in time and space, that is, anytime between any places. Ubiquity of information services makes data communication an indispensable part of information systems.

Durability. Information services have not only to do with deriving new information from older information but also act as a kind of business memory. Access to older information in the form of stored data must remain possible at any time into an unlimited future, unless and until the data is explicitly overwritten. Durability of information makes database management a second indispensable ingredient of information systems.

Interpretability. In an information system, data is exchanged across both, space due to ubiquity and time due to durability. Data carries information, but it is not information by itself. To exchange information, the sender has to encode its information as data, and the receiver reconstructs the information by interpreting the data. Any exchange should ensure, to the extent possible, that the interpretations of sender and receiver agree, that is, that meaning is preserved in space and time. This requires some common conventions, e.g., a formal framework for interpretation. Because information systems and their environment usually are only loosely coupled, the formal framework can only reflect something like a best effort. Best-effort interpretability is often called (semantic) *consistency*.

Robustness. The service must remain reliable, i.e., guarantee its functionality and qualities to any client, under all circumstances, be they errors, disruptions, failures, incursions, interferences. Robustness must always be founded on a failure model. There may be different models for different causes. For example, a service function must reach a defined state in case of failure (*failure resilience*), service functions must only interact in predefined ways if they access the same resource (*conflict resilience*), and the effect of a function must not be lost once the function came to a successful end (*function persistency*).

Security. Services must remain trustworthy, that is, show no effects beyond the guaranteed functionality and qualities, and include only the predetermined clients, in the face of failures, errors or malicious attacks.

Performance. Services must be rendered with adequate technical performance at given cost. From a client's perspective the performance manifests itself as the response time. From a whole community of clients the performance is measured as throughput.

Scalability. Modern information systems are open systems in the number of both, clients and servers. Services must not deteriorate in functionality and qualities in the face of a continuous growth of service requests from clients or other servers.

3 Service hierarchies

3.1 Divide-and-conquer

Given a requirements specification in terms of service functionality and qualities on the one hand and a set of available basic, e.g., physical resources from which to construct them on the other hand, architectural design is about solving the complex task of bridging the gap between the two. The time-proven method for doing so is divide-and conquer which recursively derives from a given task a set of more limited tasks that can be combined to realize the original task. However, this is little more than an abstract principle that still leaves open the strategy that governs the decomposition.

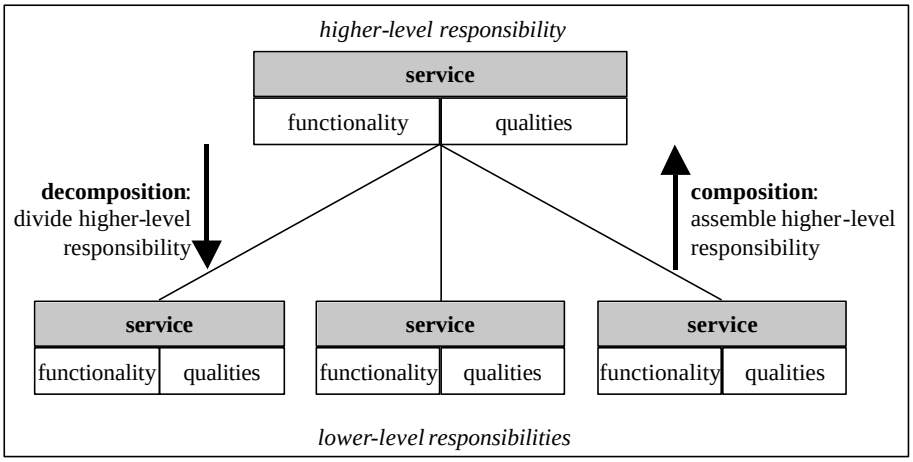

Figure 2 Divide-and-conquer for services

We look for a strategy that is well-suited to our service philosophy. Among the various strategies covered in [St02] the one to fit the service philosophy best is the *assignment of responsibilities*. In decomposing a larger task new smaller tasks are defined, that circumscribe narrower responsibilities within the original responsibility (Figure 2). If we follow Section 2.2, a responsibility no matter what its range is always defined in terms of a service functionality and a set of service qualities. Hence, the decomposition results in a hierarchy of responsibilities, i.e., services, starting from the

semantically richest though least detailed service at the root and progressing downwards to ever narrower but more detailed services. The inner nodes of the hierarchy can be interpreted as resource managers that act as both, service providers and service clients.

3.2 Design hypothesis

All we know at this point is that decomposition follows a strategy of dividing responsibilities for services. Services encompass functionality and a large number of quality-of-service (QoS) parameters. This opens up a large design space at each step. A design method deserves its name only if we impose a certain discipline that restricts the design space at each step.

The challenge now is to find a discipline that both, explains common existing architectural patterns, and systematically constructs new patterns if novel requirements arise. We claim that the service perspective has remained largely unexplored so that any discipline based on it is as yet little more than a *design hypothesis*.

Our method divides each step from one level to the next into three parts.

Functional decomposition.
This is the traditional approach. We consider service functionality as a primary criterion for decomposition. Since the original service requirements reflect the needs of the business world, the natural inclination is to use a pure top-down or *stepwise decomposition* strategy. At each design step a service functionality is given, and we must decide whether, and if so how, the functionality should be further broken up into a set of less powerful obligations and corresponding service functionalities to which some tasks can be delegated, and how these are to be combined to obtain the original functionality. However, the closer we come to the basic resources the more these will restrict our freedom of design. Consequently, at some point we may have to reverse the direction and use *stepwise composition* to construct a more powerful functionality from simpler functionalities.

Propagation of service qualities.
Consider two successive levels in the hierarchy and an assignment of QoS-parameters to the higher-level service, we now determine which service qualities should be taken care of by the services on the upper and lower levels. Three options exist for each quality. Under *exclusive control* the higher-level service takes sole responsibility, i.e., does not propagate the quality any further. Under *partial control* it shares the responsibility with some lower-level service, i.e., passes some QoS aspects along. Under *complete delegation* the higher-level service ignores the quality altogether and entirely passes it further down to a lower-level service. For partial control or complete delegation our hope is that the various qualities passed down are orthogonal and hence can be assigned to separate and largely independent resource managers.

Priority of service qualities.
Among the service qualities under exclusive or partial control, choose one as the primary quality and refine the decomposition. Our hope is that the remaining qualities exert no or only minor influences on this level, i.e., are orthogonal to the primary quality and thus can be taken care of separately.

Clearly, there are interdependencies between the three parts so that we should expect to iterate through them.

4 Testing the design hypothesis

4.1 Classical 5-layer architecture

Even though it is difficult to discern from the complex architecture of today's relational DBMS, most of them started out with an architecture that took as its reference the well-published 5-layer architecture of System R [As79, Ch81]. Up to these days the architecture is still the backbone of academic courses in database system implementation (see, e.g., [HR99]). As a first test we examine whether our design hypothesis could retroactively explain this (centralized) architecture.

4.1.1 Priority on performance

We assume that the DBMS offers all the service qualities of Section 2.3 safe ubiquity, and we ignore security for the time being. The service functionality is determined by the relational data model in its SQL appearance.

As noted in Section 2.3, durability is the raison d'être for DBMS. Durability is first of all a quality that must be guaranteed on the level of physical resources, by non-volatile storage. Let's assume that durability is delegated all the way down to this level. Even after decades durability is still served almost exclusively by magnetic disk storage. If we use processor speed as the yardstick, the overwhelming bottleneck, by six orders of magnitude, is access latency, which is composed of the movement of the mechanical access mechanism for reaching a cylinder and the rotational delay until the desired data block appears under the read/write head. Consequently, performance dwarfs all other service qualities in importance on the lowest level. Considering the size of the bottleneck and the fact that performance is also an issue for the clients, it seems to make sense to work from the hypothesis that performance is the highest-priority quality across the entire hierarchy to be constructed.

4.1.2 Playing off functionality versus performance

Since we ignore for the time being all service qualities except performance, our design hypothesis becomes somewhat simplified: There is a single top-priority quality, and because it pervades the entire hierarchy it is implemented by partial control. The challenge, then, is to find for each level a suitable benchmark against which to evaluate

performance. Such a benchmark is given by an access profile, that is a sequence of operations that reflects, e.g., average behavior or high-priority requests. We refer to such a benchmark as *data staging*.

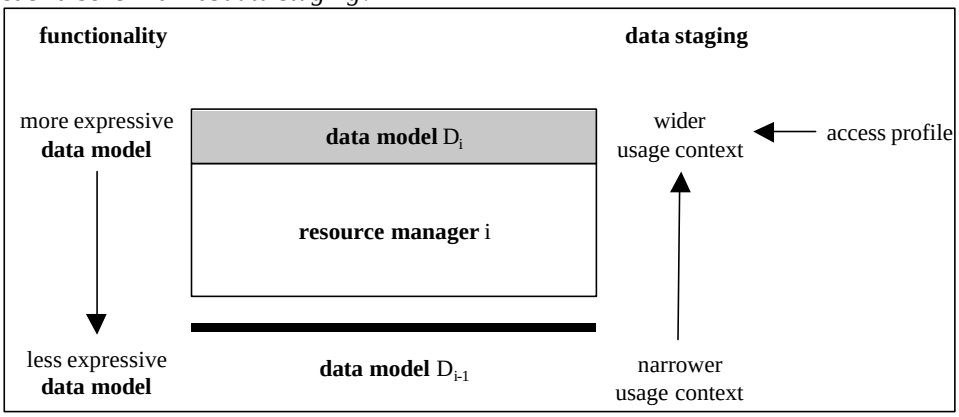

Figure 3 Balancing functionality and performance on a level

Consequently, our main objective on each level is determining a balance of functionality and data staging. As Figure 3 illustrates, the balancing takes account of a tandem of knowledge. On the way down we move from more to less expressive data models and at the same time from a wider context, i.e., more global knowledge of prospective data usage, to a narrower context with more localized knowledge of data usage. The higher we are in the hierarchy, the earlier can we predict the need for a data element. Design for performance, then, means to put the predictions to good use. Based on these abstractions we are indeed able to explain the classical architecture.

- We start with the root whose functionality is given by the relational model and SQL. The logical database structure in the form of relations is imposed by the clients. We also assume an access profile in terms of a history of operations on the logical database. We compress the access profile into an access density that expresses the probability of joint use of data elements within a given time interval. The topmost resource manager can now use the access density to rearrange the data elements into sets of jointly accessible elements. It then takes account of performance by translating queries against the relational database to those against the rearranged, internal database. The data model on this internal level could very well still be relational. But since we have to move to a less expressive data model, we leave only the structure relational but employ tuple operators rather than set operators. Consequently, the topmost resource manager also implements the relational operators by programs on sets of tuples.
- What is missing from the access density is the dynamics – which operations are applied to which data elements and in which order. Therefore, for the next lower level we compress the access profile into an access pattern that reflects the frequency and temporal distribution of the operations on data elements. There is a large number of so-called physical data structures tailored to different patterns –

take hash algorithms for associative access, list structures for sequential access, trees for combined associative and sequential access. The resource manager on this level accounts for performance by assigning suitable physical structures to the sets of the internal data model. The data model on the next lower level provides a library of physical data structures together with the operators for accessing them.

- It is not all clear how to continue from here on downwards because we have extracted all we could from the access profile. Hence we elect to change direction and start from the bottom. Given the storage devices we use physical file management as provided by operating systems. We choose a block-oriented file organization because it makes the least assumptions about subsequent use of the data and offers a homogeneous view on all devices. We use parameter settings to influence performance. The parameters concern, among others, file size and dynamic growth, block size, block placement, block addressing (virtual or physical). To lay the foundation for data staging we would like to control physical proximity: adjacent block numbering should be equivalent to minimal latency on sequential, or (in case of RAID) parallel access. The data model is defined by classical file management functions.
- The next upper level recognizes the fact that on the higher levels data staging is in terms of sets of records. It introduces its own version of sets, namely segments. These are defined on pages with a size equal to block size. Performance is controlled by the strategy that places pages in blocks. Particularly critical to performance is the assumption that record size is much lower than page size so that a page contains a fairly large number of records. Hence, under the best of circumstances a page transfer into main memory results in the transfer of a large number of jointly used records. Buffer management gives shared records a much better chance to survive in main memory. The data model on this level is terms of sets of pages and operators on these.
- This leaves just the gap to be closed between sets of records as they manifest themselves in the physical data structures, and sets of pages. Given a page, all records on the page can be accessed with main memory speed. Since each data structure reflects a particular pattern of record operations, we translate the pattern into a strategy for placing jointly used records on the same page (record clustering). The physical data resource manager places or retrieves records on or from pages, respectively.

Figure 4 summarizes the discussion.

4.1.3 Taking consistency into consideration

Data models represent generic functionalities, that is they are described by polymorphic type systems. Consequently, the managers in the service hierarchy of Section 4.1.2 deal with databases and queries generically. On the other hand, databases and queries against them must be monomorphic to be able to process them. Data models are instantiated to monomorphic type systems by specifying database schemas that are derived from the application semantics. Hence, consistency manifests itself in database schemas.

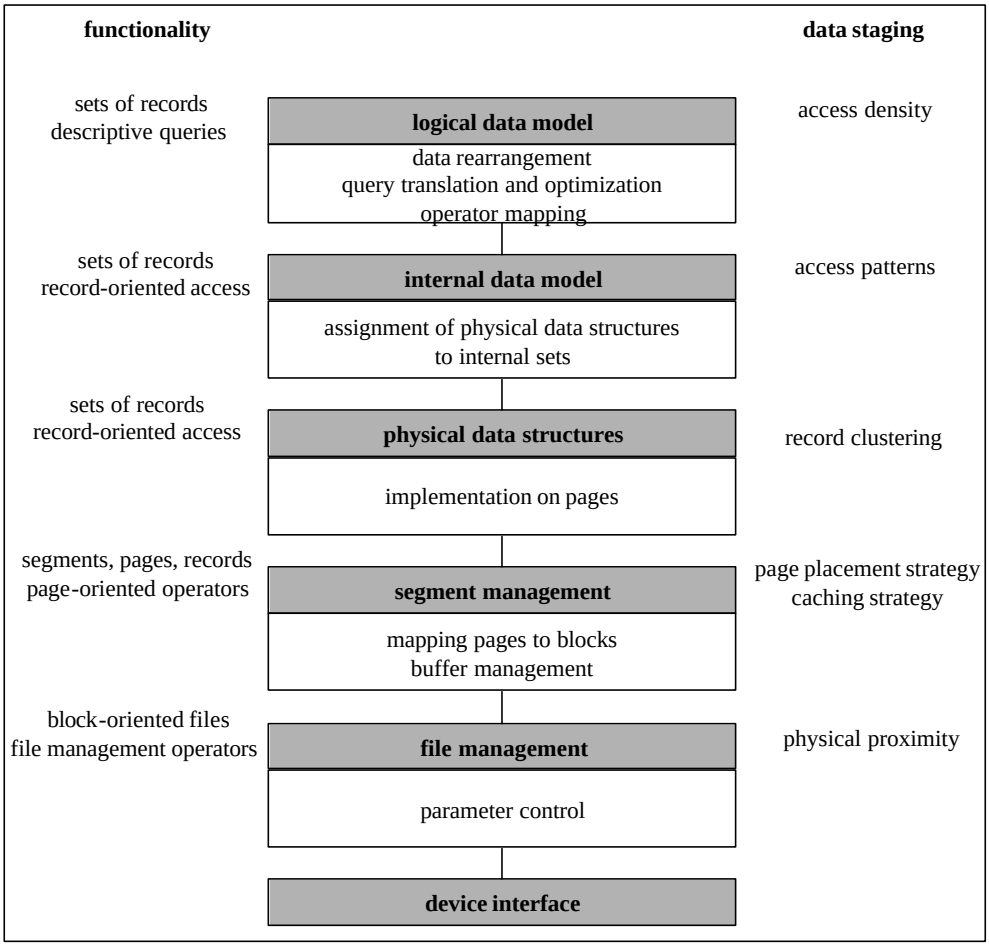

Figure 4 Reference service hierarchy for set/record-oriented database management systems

We conclude from the design process in Section 4.1.2 that only the upper two managers for logical and internal databases need a rich type system. Hence, consistency is the responsibility of just the upper two managers. Access density and patterns must be expressed in terms of the database schema to make sense.

Both managers use the schema to interpret the queries and to control the performance. Both access the schema but have no need to manage by themselves the functions for accepting, checking, storing and retrieving the schema. Nor does any of the lower managers appear to be a candidate to which to delegate this functionality. Consequently, we add a new service, meta data management, that is used by the two managers. As a service shared by two levels it seems to fall outside the hierarchy of Figure 4 (Figure 5). With the new service we may again associate service qualities such as (meta-)consistency or durability so that the design process should be repeated for the new branch.

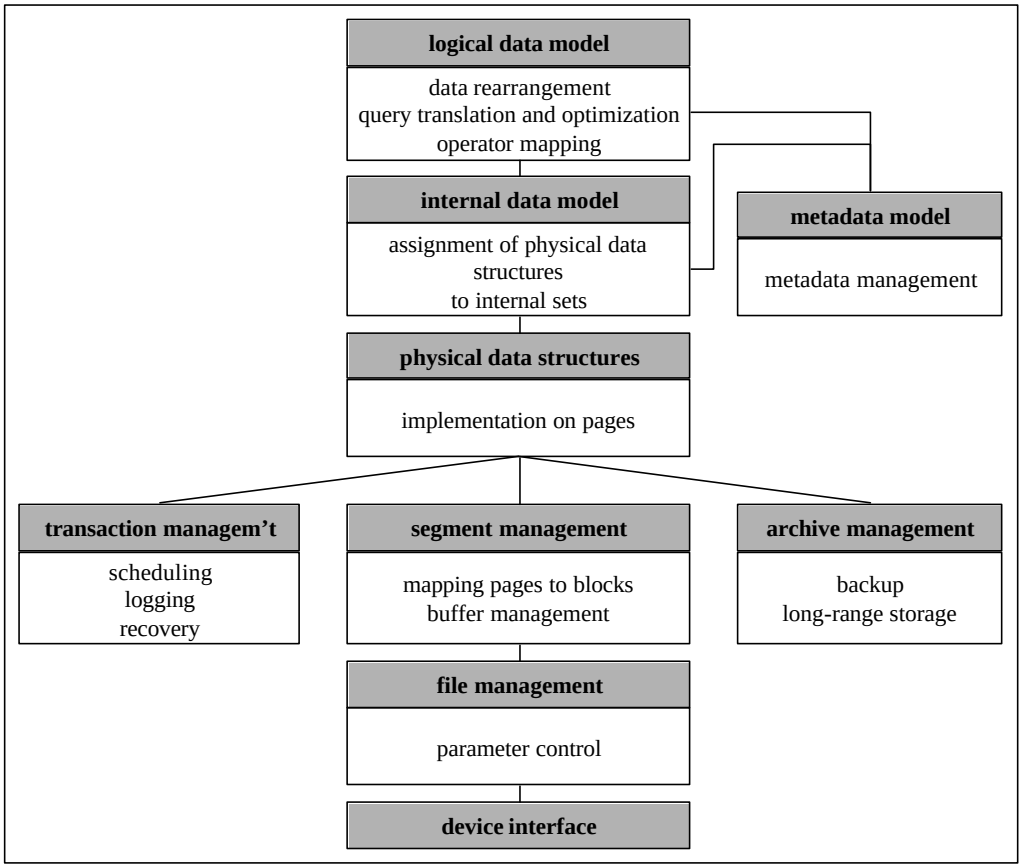

Figure 5 Augmented reference service hierarchy of Figure 4

4.1.4 Adding robustness

Consistency in terms of the database schema guarantees that the database reflects a possible state of the environment. Transactions can ensure that the database corresponds to the current state. Consequently, robustness with failure resilience, conflict resilience and function persistency is defined in terms of transactions as atomic processing units. With the exception of Weikum's multi-level transactions [We88] all reference architectures deal with transactional qualities (at least if they are ACID) on levels that are devoid of application semantics. In the design of Section 4.1.2 the proper level for transaction management would then be the segment level.

The segment level must now take care of two service qualities, performance and robustness. According to our design hypothesis we have to decide if the two are orthogonal. Intuitively one indeed considers them orthogonal because one would like to have performance even if there was no robustness, and robustness even if performance was not an issue. As a result one would split the segment level into two resource

managers, segment management proper and transaction management. Figure 5 shows the extension of the architecture. In fact, the two qualities are not entirely independent of one another so that the two managers should closely communicate with one another.

4.1.5 Adding durability

Peripheral storage may itself be subject to technical failures or external catastrophic events. Consequently, there is more to durability than what we considered in Section 4.1.2. The arguments in Section 4.1.4 can be repeated here: Durability is a generic property that ought to apply irrespective of application semantics, and it is orthogonal to performance and robustness (to the latter because of its much longer time horizon). Consequently, we add another resource manager, archive management, on the segment level (Figure 5).

4.2 Semistructured databases

Our design hypothesis held up quite well to explain the classical 5-layer architecture both in its core and its extensions. A somewhat harder test would be to try and apply the hypothesis to an area where there is less agreement as to the best reference architecture: semistructured databases or more specifically, XML databases.

4.2.1 Front-ends

For the sake of comparison with Section 4.1, let us assume that there is still consistency to be observed, that is, there is a database schema (either DTD or XML Schema). Under these circumstances we currently find two approaches. One imposes external factors, either technical such as interoperability between DBMS on the basis of XML as the data exchange format, or economical such as minimal cost of re-implementation. The result is some kind of XML front-end to a relational DBMS. The second approach builds a tailored, so-called native DBMS for XML.

In terms of our design hypothesis, one could explain the front-end as singling out the external factors as additional qualities of service that are kept under exclusive control. The XML data model significantly differs from the relational data model, though. The data structures are hierarchies rather than flat tuple sets, access is navigational by path expressions rather than set-algebraic, the nodes in the tree may have structural differences even if they satisfy the same type, and because of the document history nodes may include long texts or other media data. Consequently, the backside of the approach is that the front-end, if it deals with performance at all, does so on criteria that differ from the rest of the system.

4.2.2 Native systems

There is no reason to believe that performance plays a lesser role for (pure) XML databases. Therefore the design process of Section 4.1.2 based on seamless

performance should apply here as well, though we should expect that the differences in the data model have a significant effect.

- The root functionality is now given by XML, with the logical database structure in the form of trees and reading access by path expressions that identify subtrees. For writing access there is as yet no common standard. Some vendors prefer simple delete/write for modifications, or experiment with XSLT. Consequently, there is no clear way to separate access density and patterns – both take the form of navigated trees. Density is complicated by the fact that nodes have large structural variances as to number of attributes, cardinality of same-tagged successors and size of attached media data, and patterns allow various selection choices due to the structural variances. Indeed, native XML database products seem to collapse the two upper resource managers of the relational system into a single, fairly complex manager.
- The next lower resource manager is now something akin to the physical data structures. Basically one would expect three kinds of data structures: Subtrees of XML structures, index structures for navigating through XML structures, and media data of possibly large size.
- Each of these physical structures can then be optimized with regard to performance. Index structures come closest to the relational situation and may thus be realized on the segment level similar to Section 4.1.2. Subtrees may vary widely in size so that either large pages sizes must be chosen or subtrees may span a number of pages. Both require solutions that differ somewhat from that typical for relations, so that segment management becomes definitely more complicated. Media data would extend across many pages, moreover classical buffer caching would make little sense for them. They would, therefore, directly draw on the services of file management.

Figure 6 summarizes the discussion. Viewed superficially the architecture looks simpler than for the relational case. In fact, though, it is just less structured because it seems more difficult to decompose the services. In the end each resource manager in the architecture is more complicated than in the relational case. We note in passing that the problem of media data is also known as well for relational systems where large binary fields are used for the purpose.

5 Putting the design hypothesis to work

Section 4 seems to bear out the validity of our design hypothesis. But does it really? Or – so one might suspect – did we just formulate the hypothesis to fit the well-established architectures? Better proof would be to find constructive solutions to some novel situations.

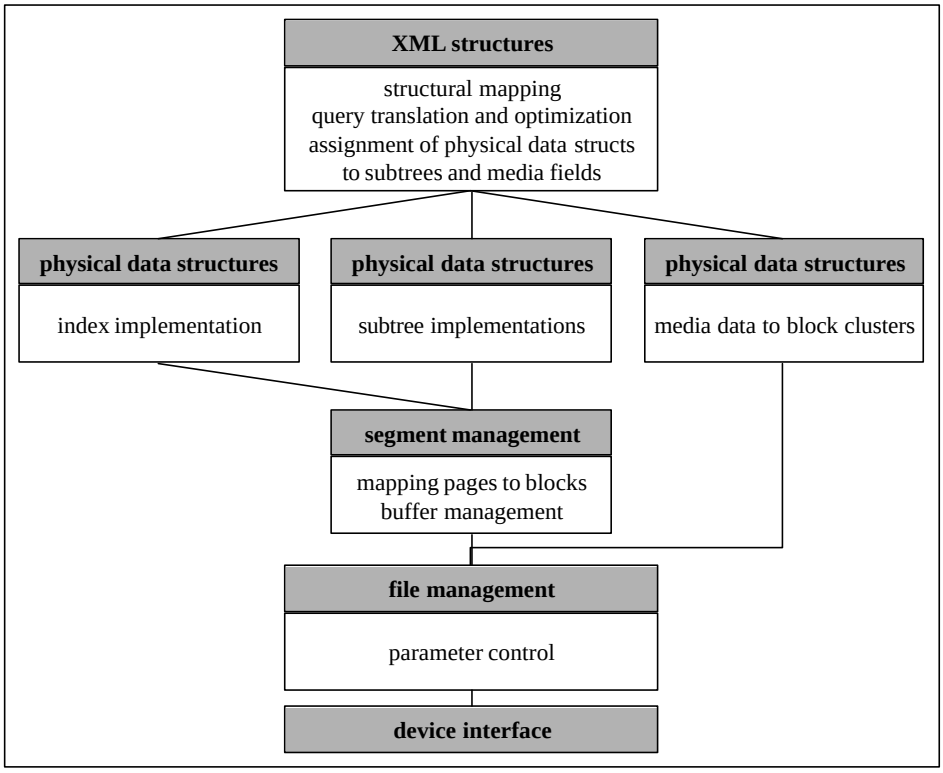

Figure 6 Service hierarchy for semistructured database systems

5.1 Hippocratic databases

In a recent paper [Ag02] Rakesh Agrawal et al. introduce the concept of *Hippocratic databases* for database systems that should enforce all political and societal rules to protect the privacy of data. The authors translate the requirement into the service quality of *purpose*: Any acquisition of data, their durability over a time span, and any queries on them have to serve a specific purpose. A purpose should be formulated in a precise fashion, all persons affected should agree, and there should be no access to the data without the purpose. The authors then go on to propose an architecture. The first impression is that the authors pretty much stick to the service hierarchy of Section 4.1 and just augment it by additional, albeit complex components. In other words, the authors do not question the prevalence of performance.

What would have happened if they had done so? If they had made purpose their prime quality? Let's try for a hypothetical answer (Figure 7). Suppose that we query the database in the usual way, except that the query is accompanied by its purpose. We expect that the result is those data whose purpose is compatible with the purpose of the

query. All other data remain invisible to the client, i.e., become visible neither by accident nor by intent.

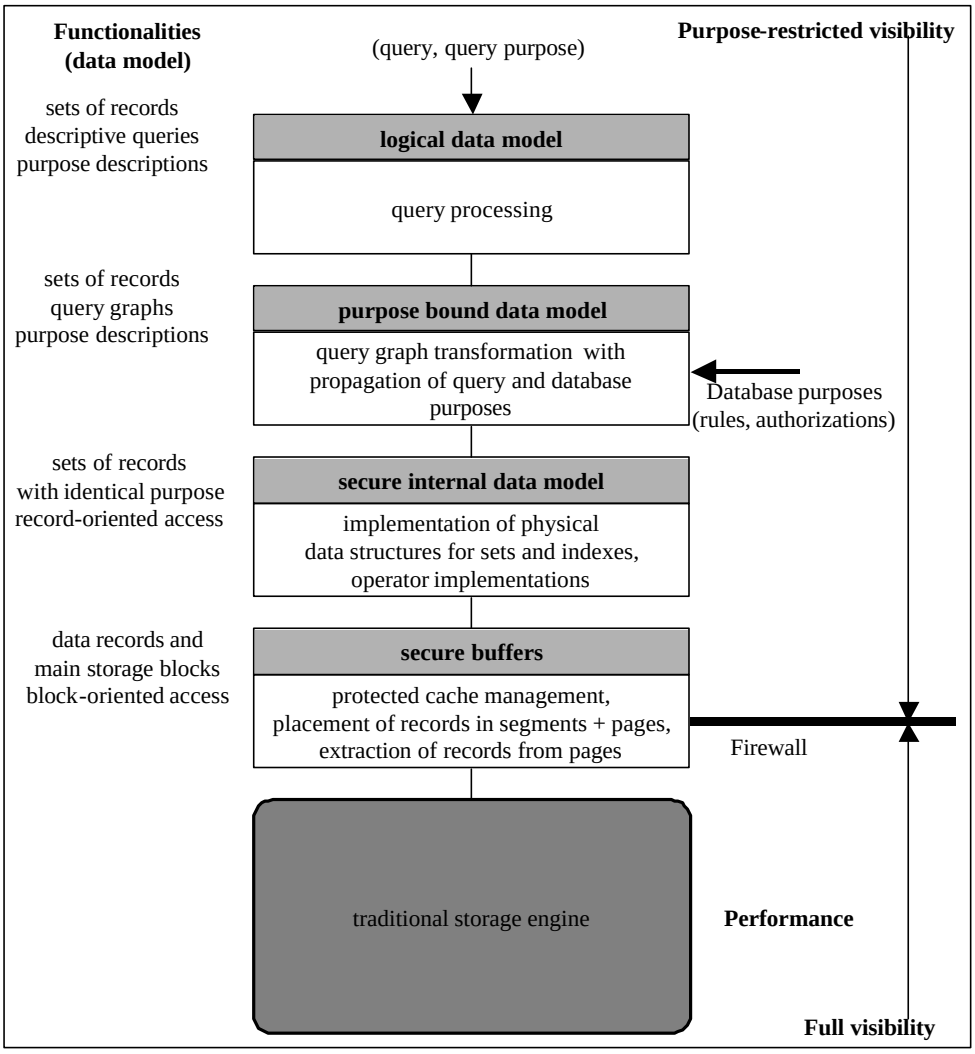

Figure 7 Hypothetical service hierarchy for Hippocratic databases

Notice that the purposes of the database are not open to inspection by regular clients. Consequently, they can only be seen from the second level on downwards. The uppermost level will confine itself to standard query processing into, e.g., a query graph together with information on the query purpose. This is input to the second level together with the rules, e.g., authorization rules, on database purposes. This allows to compute the visible nodes in the query graph. The third level sets up the physical data

structures for the visible data and computes their materializations. In other words, this level will separate the visible data from the invisible ones and reproduce the former in a separate main storage area. The next lower level is the most critical one because it must do the physical separation, for instance in a well-protected buffer area, and thus is something akin to an internal firewall. Only below can we proceed with the traditional storage engine from the level of segment management on downwards. And only from there on replaces performance purpose as the prime quality.

There is an apparent conclusion: Privacy protection in DBMS comes at the price of lower performance.

5.2 Incorruptible databases

While Hippocratic databases restrain access according to purpose no matter what the intent, *incorruptible databases* restrain changes no matter what the purpose. More precisely, the goal of incorruptible databases is to allow only those updates that satisfy certain rules, i.e., that make sense or seem plausible. Incorruptibility can be seen as a special kind of consistency. Consistency thus becomes the prevailing quality of service.

Plausibility can be tested in various ways. For example, one may employ statistical and data mining techniques to detect outliers, or one may formulate somehow restricted first-order logic formulae to check new input in the light of absolute constraints, the current database state or the previous history of database states. The second approach has a lot of similarity to deductive databases. We try to follow this approach.

Deductive constraint checking is known to be compute-bound rather than I/O-bound. Clearly then, performance is again the quality of priority. But in contrast to Section 4 where it was due to durability it is now due to consistency. Moreover, whereas in Section 4 we assume queries to be spontaneous or any of a large number of parameterized queries, consistency constraints are stable over a long period of time so that preprocessing makes sense. Further, a large number of constraints must be observed at any one time so that optimization techniques can employ additional techniques such as common subexpressions [He96]. Consequently, the service hierarchy splits at the top into a preprocessing ("compile") service and a query-and-update ("just-in-time") service. The later does the just-in-time rule processing employing the results of the precompiled checking modules. Deductive databases usually employ relational databases that have been augmented by special algebraic operators. Both precompilation and just-in-time optimization should generate queries that fetch into a special main memory cache those database portions to be checked, but that are formulated in such a way as to delegate the optimization with respect to data staging to traditional levels of a DBMS. Figure 8 summarizes the hypothetical solution.

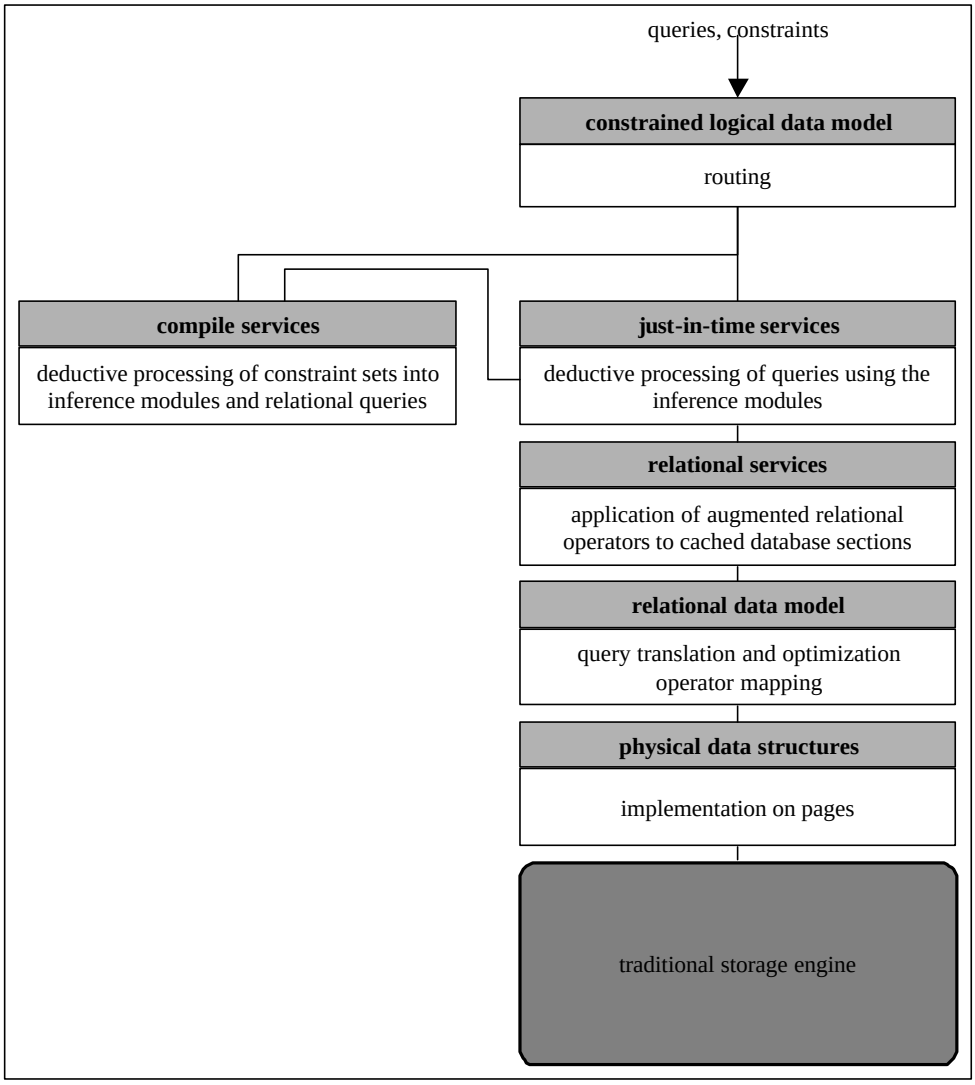

Figure 8 Hypothetical service hierarchy for incorruptible databases

6 Tight and loose coupling

6.1 Layered and component architectures

The reader may have noticed that what we have constructed so far were service hierarchies. These are not yet architectures but just the preliminary steps towards an architecture.

System architectures fall into two broad categories, *layered architectures* and *component architectures*. Two successive levels in a service hierarchy form successive layers if there is a close coupling between the two. If the coupling is loose, the levels form separate components.

Intuitively speaking, close coupling reflects a higher degree of interdependence, loose coupling a lesser degree. In view of the broad meaning of interdependence it will be difficult to find a single factor that determines the strength of coupling, and therefore to give a precise formulation. Instead we take a phenomenological approach.

- One conceivable criterion for loose coupling is that the two services either share no resources, or do so free of conflicts. An example is services that run on different computers or in different processes.
- One may consider as a case of loose coupling if two levels are separated by exclusive control or by complete delegation, that is, if they attend to different service qualities. Hence, the two share no responsibilities, and they can be expected to pursue independent strategies and thus to achieve a fair degree of autonomy.
- Even in case of partial control there may be a case for loose coupling if satisfying the quality is achieved by moving in different directions, indicating that different conditions come into play. Just remember the reversal of direction during the design of the classical 5-layer architecture in Section 4.1.2 due to the disappearance of application semantics.
- From software engineering it is well-known that the traffic density between two resource managers, e.g., the number of service calls or messages per time unit, determines the needed strength of the coupling. Higher densities require close coupling to avoid a decline in performance.
- Finally, a service may be of general value to different service hierarchies and thus be called from various clients. In this case close coupling would give an unfair advantage to just one client.

These criteria may occur in combination, and are not even completely orthogonal. Consequently, in a given situation there may still be a choice of whether to translate a hierarchy into a layered or a component architecture.

6.2 Service hierarchies revisited

One can easily demonstrate that with the four criteria above there is more than one architectural solution for the hierarchy of Section 4.1 (Figure 5). Fairly straightforward is the case for metadata and archive management. Both have their own resources, the top two levels of the hierarchy usually call on metadata only during certain start-up phases, and archive management kicks in only at larger intervals. Hence, both are natural components.

The boundary between physical data structures reflects a reversal of direction. Therefore one may opt for two components comprising the levels above and below, respectively. On the other hand, the two levels share the buffers located in the segment level. As a consequence, there is also a high traffic density between the two by placing and accessing records. Further traffic arises from resolving conflicts between the two. And indeed, during the long history of database systems both, complete layering and separation into two components, can be found, although the latter has mostly been confined to prototypes.

Similar considerations hold for transaction management. Transaction management has (almost) exclusive control over robustness, suggesting a component solution. However, usually transaction management carries some responsibility for performance, indicated by sharing the buffer with segment management. Traditionally, the performance argument won the day, and transaction management migrated into segment management. An exception is the QoS of conflict resilience which has lately even shown a tendency to move outside the DBMS.

Figure 9 illustrates the extreme of full componentization.

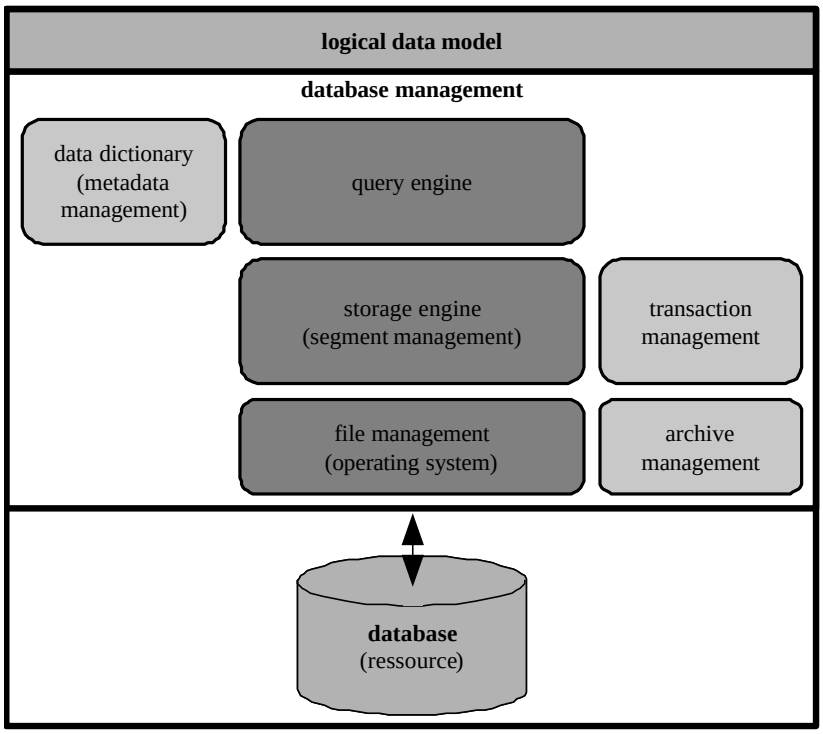

Figure 9 Component architecture for the service hierarchy of Figure 5

One may apply a similar design strategy to the other service hierarchies. For example, for the hierarchy of Section 4.2 (Figure 6) there is a somewhat stronger though still not overwhelming argument for providing separate storage engines. In Figure 7 the firewall mandates loose coupling, with a separate storage engine and perhaps part of the secure buffers level placed in a separate component. In Figure 8 the compile services qualify as a component, and the boundary between relational services and relational data model has many traits of loose coupling.

We conclude that from a programming-in-the-very-large standpoint, if given a service hierarchy, components determine a coarse-granular system architecture and layers the fine-granular architecture within a component.

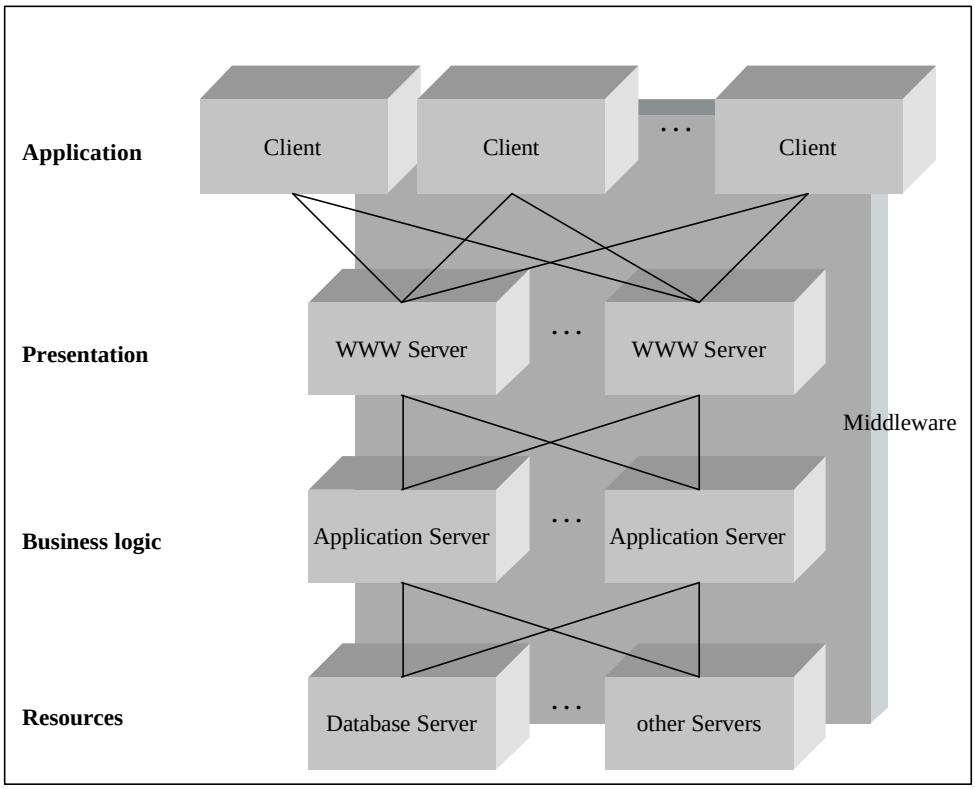

Figure 10 Four-tier architecture in a multidimensional framework

7 Coarse-granular component architectures

7.1 Tiered and multi-dimensional architectures

Information systems are more than just database systems. Indeed today, database systems are almost entirely hidden behind middleware and application servers. On the other hand, modern information systems are still service providers so that our design approach should carry over to these much coarser architectures.

As just mentioned, coarse architectures should be component architectures. The services may still be part of a service hierarchy. If this is the case we speak of a *tiered architecture*. A modern example is the 4-tier architecture with presentation clients, presentation servers, application servers and data managers (Figure 10).

The 4-tier architecture needs an infrastructure for communication and interoperability, commonly referred to as middleware. Figure 10 illustrates how the middleware as a component is called from each level in the service hierarchy so that it seems to pervade the entire hierarchy. One may visualize such an arrangement as the hierarchy forming one dimension and the common component (which in all likelihood is itself a service hierarchy) defining a second dimension. We refer to such an arrangement as a *multi-dimensional architecture*.

To illustrate the principle, consider as a very simple example Figure 11 [Ab03]. It shows a database server together with some application client. Both communicate across a transmission network with its own layered architecture (in our case the bottom four OSI/ISO levels). Space defines the third dimension.

7.2 Design hypothesis revisited

Take the architecture of Figure 10. Three of the four component types exhibit a relatively narrow and standardized functionality: client, Web server and database server. The specifics of an application – the business logic in the form of service functionality and QoS parameters – are concentrated in just one component, the application server.

Compared to Sections 4 through 6 the situation is now different: The component architecture is not of our own design but is being imposed – for good software engineering reasons. All we can hope for is that each component does its best to meet its QoS parameters. But as we all know, a set of local optima does not necessarily result in the global optimum. To approach the latter, we modify our design hypothesis as follows.

1. Assign service qualities to each component.
2. If the component can be influenced, develop it according to the design hypothesis of Section 3.2.
3. For each quality shared between adjacent components and arranged in order of priority, increase by technical means the strength of the coupling.

4. Re-evaluate and adjust the adjacent components in the light of the coupling technique.

In essence, because we can no longer start with a clean slate we employ a heuristic that first develops local optima and then, by iteration, tries to approach the global optimum.

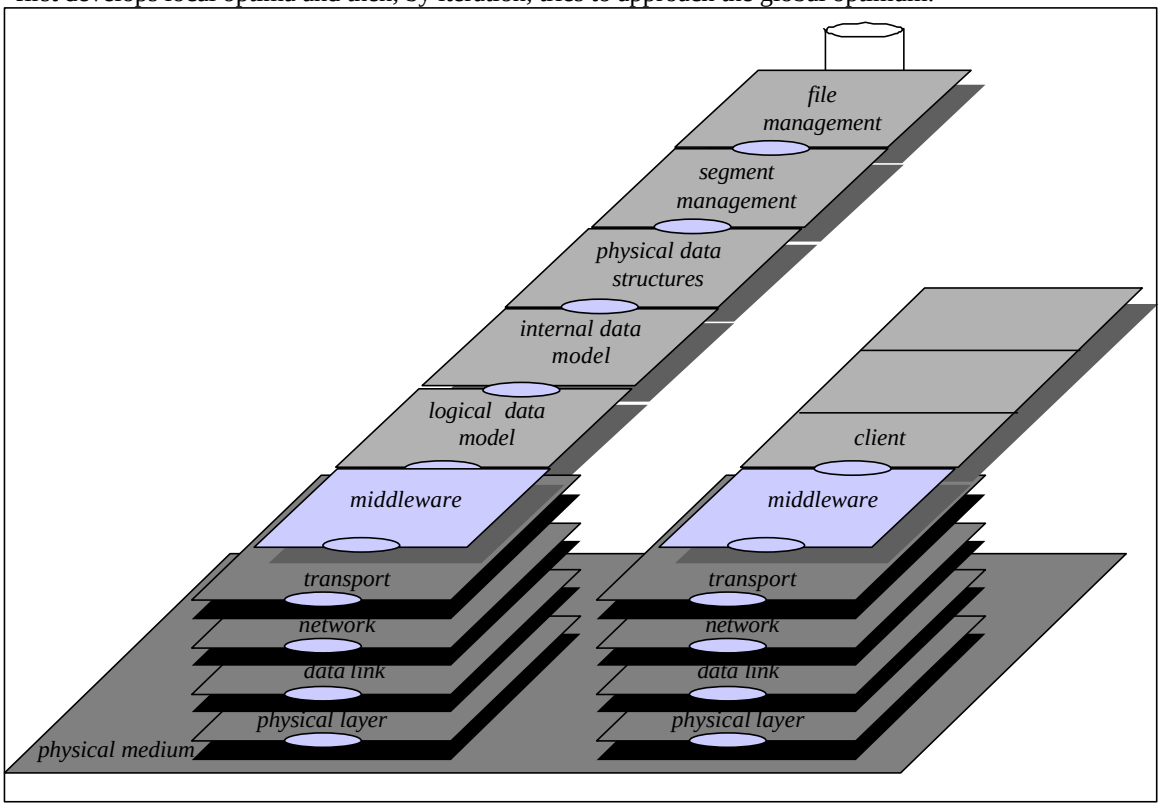

Figure 11 Multidimensional architecture linking a client to a database system via data communication

7.3 The design hypothesis at work

7.3.1 Priority on performance

Suppose that as in Section 4 performance is our highest-priority quality. Consequently, our design goal is to minimize the performance loss across the coupling between adjacent components. The basic principle should remain the same as in Section 4.1.2: Employ *data staging* to take the access profile into account. In the centralized architecture of Section 4.1.2 and with page size exceeding record size by a large factor a single buffer in the storage engine proved sufficient. For loose coupling, though, we will have to replicate and rearrange data as we move upwards in the hierarchy and consider progressively wider context, as we already did in some sense in Figure 7. In

other words, data staging via buffering, or as it is more generally called, by *caching* will be needed for each component coupling. Figure 12 reflects the principle.

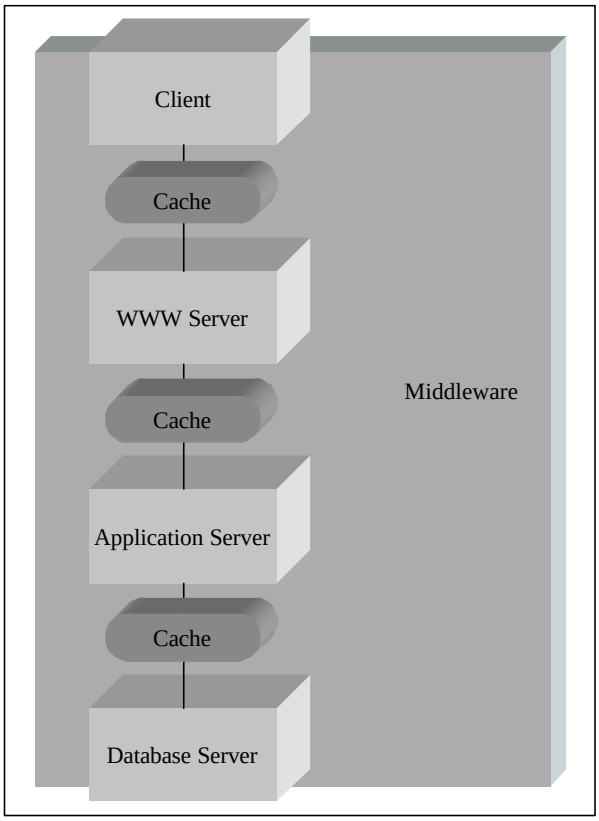

Figure 12 Performance control in a four-tier architecture

Translating the conceptual architecture into a physical architecture is by no means trivial, though. Several options exist for a cache.

- Separate component. The major advantage is that the autonomy of each component is maintained. No intrusion is necessary, each participating component remains as it is. On the other hand, all communication with the cache has to go through the middleware, adding considerable overhead and, hence, reducing the benefits of caching.
- Integration into the middleware. As before, the autonomy of each component is maintained. Data communication overhead is reduced because the buffer is part of the middleware. Because caching now becomes a generic middleware service one may encounter difficulties in trying to tailor it to the data staging needs of a specific connection.

- Integration into a participating component. We have to decide which component to choose and, hence, to modify for the purpose. If we employ the context argument then this should be the component in the hierarchy which has the best knowledge of the context.
- Migration. The cache is attached, i.e., physically coupled to the middleware or one component (edge-of-net or edge-of-server). The advantages of edge-of-server are the same as for a separate solution, but with lesser data communication overhead. Somewhat more coordination overhead arises between component and cache as compared to full integration.

These options give rise to a wide range of combinations and thus to a large optimization space, as pointed out by Mohan [Mo01]. Examples abound. Caches for the connection between application server and database server, so-called application data caches, can be edge-of-database-server if they materialize database queries, provided the query load is fairly homogeneous. They are edge-of-application-server if they materialize local database queries [Lu02] or implement mapping functionality, e.g., object-relational mapping as in J2EE container-based persistency. Integrated solutions are those where application data is cached through runtime objects designed by the application developer, as in J2EE bean-based persistency.

At the upper end, Web page caching usually is integrated into the WWW server, i.e., not directly accessible by the client through a separate service. Beyond the simple task of caching static pages, there are many approaches for caching dynamically generated Web pages. Often, edge-of-net caches known as Web proxy caches are added to the connection between WWW client and WWW server.

Novel challenges arise for the connection between WWW server and application server. Whereas database servers and Web servers offer a fairly narrow generic functionality, application servers may offer extensive and complex services that vary widely across servers. Consequently, all communication is via service function calls, e.g., in case of object-orientation via method calls. Since there is little likelihood that method calls are shared across different Web servers nor across different applications on the same Web server, the most natural solution seems one of cache integration into the Web server (method cache) [PW02].

7.3.2 The curse of consistency

Our notion of consistency refers to interpretability of the database as a possible state of the environment. This leaves consistency in the hands of the database server in our tiered architecture. If the database schema remains the same across all servers the meaning of the data is preserved across them. Conversely, if mappings take place in the caches the consistency may become endangered. In general, mappings are no longer generic so that an orthogonal solution along the lines of Section 4.1.3 seems more difficult to achieve.

A stricter notion associates consistency with the current state of the environment and enforces it by transactions. As data move upwards from peripheral storage across the

caches where they become replicated and rearranged according to the staging strategy, they become progressively more removed from the database state. Just consider that changes on the upper tiers propagate downwards with certain delays. The effect is known as the *cache coherency* or cache invalidation problem. If we followed the orthogonal approach of Section 4.1.4 we would have to add a global transaction manager to Figure 12 as a new, separate component. Indeed, such a solution is part of distributed systems or of middleware systems. On the downside, distributed transactions have a negative effect on both, performance and component autonomy.

If one is willing to deviate from strict consistency to a certain degree, solutions that control the propagation of changes downwards to the database server and from there upwards to other servers come into play. Examples are push versus pull strategies, synchronous versus asynchronous propagation, automatic refresh. Migrated caches seem to offer advantages over integrated caches because one may specify and enforce strategies without knowledge of and effects on the inner working of the servers.

A somewhat bitter conclusion is that in tiered architectures, as opposed to layered architectures, QoS parameters show considerable interdependence, i.e., less orthogonality. The optimum across all parameters is much more tedious to find and requires repeated iterations.

7.3.3 Adding robustness

In Section 4.1.4 robustness was defined in terms of transactions. As such it depends on strict consistency. Strict consistency, as we just noted, must be enforced by distributed and perhaps longer-duration transactions. As before, if one is willing to relax robustness and thus can do without a global transaction manager, local techniques for non-repudiation or irrevocability as in e-commerce take the place of strict consistency. On the other hand, since consistency relates solely to the database server, the caches may serve as a log for semantically richer transactions.

We observe again that robustness can not always be treated as an orthogonal quality because each connection between components may be individually affected.

7.3.4 Privacy protection

In Section 5.1 we viewed privacy protection through the concept of purpose and offered a solution for the database server that ensured that only those data were accessible whose purpose fitted the purpose of the query. Let's continue the speculation from there. We note that in the hypothetical architecture of Figure 7 data had already to be replicated on the upper layers. Hence, the caches of Figure 12 seem to be ideally suited to carry the quality of purpose up to higher tiers as purpose becomes redefined in a broader context. At the same time, though, Figure 7 introduced a firewall to separate the single query purpose from the data with their many purposes. Likewise, each connection would have to include a firewall to separate purposes.

8 Conclusion

Did we meet our self-imposed challenge to demonstrate that architectural design can be treated with some scientific rigor, that it can be made to follow a methodology that can be organized around well-defined criteria? We must leave it up to the reader to decide. By developing a design hypothesis and applying it mostly to established architectures a posteriori, we have so far only circumstantial evidence that the methodology is up to the task. Part of the evidence is that by changing the weight of the design criteria novel architectures may evolve. Even more striking is the fact that methodical design may reveal larger design spaces than initially conceived.

The design hypothesis itself revolves around the concept of service and service hierarchies. Services are described in terms of functionality and qualities. The hypothesis attempts to construct service hierarchies by decomposing service functionality under the guidance of a service quality chosen as the prime quality. The expectation is that service qualities are sufficiently orthogonal so that others can be taken care of simply by local additions or separate components. We could demonstrate that the method holds up reasonably well for layered architectures.

For component architectures where service qualities can only be controlled by proper choice of components and by regulating the connections between them we had to modify the hypothesis. We demonstrated why we suspect that with the smaller range of options we have to sacrifice the orthogonality of qualities, resulting in more iterations to arrive at a design that meets given criteria. Nonetheless, even in this situation the service philosophy seems to provide a suitable framework.

References

[Ab03] Abeck, S., Lockemann, P.C., Schiller, J., Seitz, J.: Verteilte Informationssysteme – Integration von Datenübertragungstechnik und Datenbanktechnik. dpunkt.verlag. 2003 (in German)

[Ag02] Agrawal, R.; Kiernan, J.; Srikant, R.; Xu, Y.: Hippocratic databases. Proc. 28th Intnl. VLDB Conference. 2002

[As79] Astrahan, M.M., et al.: System R: A relational database management system. IEEE Computer. 12:5. 1979. 42-48

[Ch81] Chamberlin, D.D., et al.: A history and evaluation of System R. Comm. ACM 24:10. 1981. 632-646

[He96] Herzog, U.: Effiziente Konsistenzprüfung in Datenbanksystemen. Infix. 1996 (in German)

[HR99] Härder, T.; Rahm, E.: Datenbanksysteme: Konzepte und Techniken der Implementierung. Springer, 1999 (in German)

[Lu02] Luo, Q.; Krishnamurthy, S.; Mohan, C.; Pirahesh, H.; Woo, H.; Lindsay, B.G.; Naughton, J.F.: Middle-Tier Database Caching for e-Business. Proc. ACM SIGMOD Conference. 2002. 600-611

[Mo01] Mohan, C.: Caching Technologies for Web Applications. Tutorial, VLDB 2001. http://www.almaden.ibm/u/mohan/Caching_VLDB2001.pdf

[PW02] Pfeifer, D.; Wu, Z.: A transparent client-side caching approach for application server systems. Submitted for publication. 2002

[St02] Starke, G.: Effektive Software-Architekturen – Ein praktischer Leitfaden. Carl Hanser. 2002 (in German)

[We88] Weikum, G.: Transaktionen in Datenbanksystemen. Addison-Wesley. 1988 (in German)