Database Systems Architecture: A Study in Factor-Driven Software System Design

Peter C. Lockemann

Fakultät für Informatik, Universität Karlsruhe
Postfach 6980, 76128 Karlsruhe, Germany
lockeman@ira.uka.de

Abstract. The architecture of a software system is a high-level description of the major system components, their interconnections and their interactions. The main hypothesis underlying this paper is that architectural design plays *the* strategic role in identifying, articulating, and then reconciling the desirable features with the unavoidable constraints under which a system must be developed and will operate. The hypothesis results in a two-phase design philosophy and methodology. During the first phase, the desirable features as well as the constraints are identified. The second phase is a decision process with features and constraints as the driving factors, and tradeoffs contingent on a value system that will always include subjective elements. It is of course impossible to validate the hypothesis in full generality. Instead, we restrict ourselves to an analysis – much of it retrospective – of architectures of database management systems in networks. The analysis demonstrates that the most challenging part of architectural design is to identify – very much in the abstract – those features that promise to have *the* major impact on the architecture. Further, by separating the features into two classes, a primary class with all those features that dominate the design, and a second class with those features that can then be treated orthogonally, the complexity of the design task is reduced.

1 Hypothesis

The architecture of a software system is – much in the tradition of classical systems analysis – a description of the major system components, their interconnections and their interactions. The description is on a high-level: Major features are identified, but little attention is as yet given to the details of ultimate implementation. Or in other words, developing a system architecture is "programming-in-the-very-large".

The main hypothesis underlying this paper is that architectural design is a vital first step in the development of software systems. We claim that architectural design plays *the* strategic role in identifying, articulating, and then reconciling the desirable features with the unavoidable constraints – technical, financial and personnel – under which a system must be developed and will operate. In a nutshell, architectural design is the means for explicating the major conflicts, and

K.R. Dittrich, A. Geppert, M.C. Norrie (Eds.): CAiSE 2001, LNCS 2068, pp. 13–35, 2001.

for deciding and documenting the necessary tradeoffs on a strategic level, and thus at a time when no major implementation efforts and expenses have as yet occurred. This may sound platitudinous to most, but in practice all too many flaws or failures in business systems can be traced back to the lack of an explicit architectural design and implicit mistakes when viewed from an architectural perspective.

Our hypothesis drives a two-phase design philosophy and methodology. During the first phase, the desirable features as well as the constraints are identified. During the second phase, tradeoffs are determined that preserve as many of the features as possible while minimizing the effects of the constraints. The second phase clearly is a decision process with features and constraints as the driving factors, and tradeoffs contingent on a value system that will always include subjective elements.

A scientifically rigorous approach to verifying the hypothesis would require us to set up two development teams, supply them with the same system specifications, have them follow different design strategies of which only one is strictly architecture-based, and compare the results. Moreover, such an approach would have to cover a sufficiently wide spectrum of software systems. Obviously, all this is entirely impractical. The approach to verification we take in this paper is more circumstantial, then. For one, we concentrate on a few types of database system architectures as they may appear in distributed information systems. Second, much of our analysis is retrospective. We examine system architectures that have found wide acceptance, and try to reinterpret them in the light of factors we consider particularly critical.

2 Resource Managers in Distributed Information Systems

2.1 Shared Resources and Services

Distributed information systems are a reflection of modern distributed organizations – business, administration or service industry. Today, business processes are viewed as the central concept for organizing the way business is done. Designing the business processes is, therefore, considered a major challenge. Requirements for distributed information systems should be a major outcome of the design.

Information exchange is a vital part of business processes. Business processes operate across geographical distances, often on a worldwide scale, and they utilize corporate memory in the form of huge data repositories. Consequently, we limit ourselves to a view of distributed information systems that concentrates on those features that support information exchange. Since information exchange has a *spatial* and a *temporal* aspect, distributed information systems are expected to overcome temporal and spatial distances. This gives a first – trivial – architectural criterion, the separation of the two aspects into data communications systems and database management systems.

More abstractly, from the perspective of the business process an information system can be viewed as a set of information *resources* together with a suitable (*resource*) *manager* each. Spatial exchange is provided by data communication

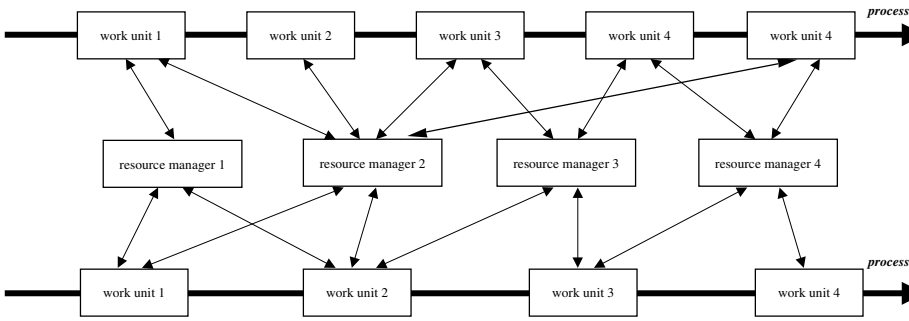

Fig. 1. Business processes and resource managers

managers, with local and global networks as their resources. Temporal exchange is supplied by database managers, with main and peripheral store as their physical resources and databases as their logical resources. The spectrum of assistance a resource manager offers to its customers is referred to as its *service*. Business processes, then, call upon the communications and database management services of a distributed information system.

Figure 1 illustrates the basic framework. A business process consists of a collection of work units. Each unit draws on the services of one or more resource managers, and different units may address the same manager. Assume for simplicity that each business process is totally ordered. Within a business process, then, resources are shared in temporal order. However, in general a large number of business processes – within the same enterprise or across different enterprises – take place in parallel.

From a service perspective, we refer to the resource managers as the *service providers* and the work units as the *service clients*.

2.2 Service Features

From an abstract perspective client and provider enter into a contract, a so-called *service agreement*. Crudely speaking, an agreement deals with two aspects: *What* is to be performed, and *how well* it is to be performed. For service features that make up the first aspect the provider is expected to give *absolute* guarantees, whereas for features within the second aspect *graded* guarantees can be negotiated. We refer to features of the first kind as *service functionality* and of the second kind as *service quality*. Ideally, service functionality would correspond to the desirable features and service quality to the constraints.

To start with the service functionality of distributed information systems, the features seem to fall into four broad categories.

1. *Utility*. This is the raison d'être for the agreement. It describes the collection of functions through which the client initiates spatial and temporal data exchange.

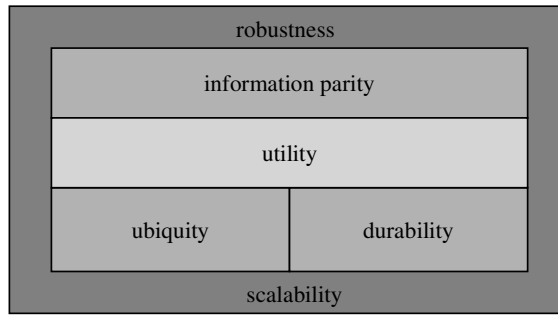

Fig. 2. Service functionality and qualities in distributed information systems

2. *Ubiquity.* Data exchange should be possible between any pair of clients at any time at any place. Data access should be possible for any client at any time from any place.
3. *Durability.* Access to stored data – unless explicitly overwritten – must remain possible at any time in an unlimited future.
4. *Information parity.* 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. This requires some common conventions, e.g., a formal framework for interpretation. The requirement is more stringent for temporal exchange than for spatial exchange because the former does not offer an opportunity for direct communication to clear up any misunderstandings. Rather, in temporal exchange the conventions must be made known to the service provider to ensure that the interpretation remains the same on data generation and access.

For distributed information systems, service quality has two major aspects.

1. *Robustness.* The service must remain reliable under any circumstances, be they errors, disruptions, failures, incursions, interferences. Robustness must always be founded on a *failure model*.
2. *Scalability.* The service must tolerate a continuous growth of service requests, both for data transmission and data storage or retrieval.

Figure 2 illustrates how the service features interact. Ubiquity is primarily the responsibility of data communications, durability of database management. All other features are the shared responsibility of both.

As seen from the contract angle, architectural design is a parallel effort to contract negotiation. Given the service features, systems designers determine how they affect each other under further constraining factors such as limitations of physical resources. They decide how they may be traded against each other and, ultimately, whether the deal can be closed or terms must be renegotiated.

2.3 Service Dynamics

From Fig. 1 we observe three kinds of relationships in an information system.

1. *Client-provider.* Clients issue *service requests* to a provider by calling a service function. The provider autonomously fills the request and returns the result to the client. Client and provider run in separate processes, the communication may be synchronous or asynchronous.
2. *Client-client.* By sharing resources, business processes, through their work units, may interact. If the interaction is wanted because the processes pursue a common objective we have a case of *cooperation.* If it is unwanted we have a case of *conflict* between the processes.
3. *Provider-provider.* A business process may call upon the services of a number of providers. To ensure that the process reaches its final objective the providers involved must *coordinate* themselves.

2.4 Refining the Hypothesis

Chapter 2 provides us with a general framework for expressing the external factors that govern architectural design. Altogether we isolated six features, too many to be considered all at once. Hence, we refine our hypothesis to one that assumes that some features exert more influence than others. The major features are used to develop a gross architecture. A measure of correct choice would be that the other features affect only a single component of the gross architecture, or add a single component to it. We refer to this property as *design orthogonality.*

The ensuing four chapters will test the hypothesis in retrospective for various database system architectures. Some of them (Chaps. 3 and 6) are widely accepted, others (Chaps. 4 and 5) are less so that this paper could even be regarded as a – modest – original contribution.

3 Database Management Systems Reference Architecture

3.1 Service Features

Our focus is database management systems (DBMS). To indicate the focus, we use a specialized terminology for the features.

1. *Data model.* A data model expresses DBMS utility. The utility is generic: Due to the huge investment that goes into the development, DBMS must be capable of supporting a large and broad market of applications. As such a data model provides a collection of primitive state spaces and transition operators, and additional constructors that allow these to be grouped into more complex state spaces and transition procedures, respectively. More formally speaking, a data model can be compared to a polymorphic type system. Operators and procedures correspond to the service functions that may be called by clients. Scalability has a functional counterpart in a constructor for dynamic sets of record-structured elements.

2. *Consistency.* Given a state of the business world, the goal is to have the database reflect a suitable abstraction of this state (the miniworld) in an up-to-date version. Information parity is thus refined to a notion of restricting the content and evolution of the data store to a well-defined set of states and state transitions considered meaningful. Each state of the database is to be interpreted as a particular state of affairs in the miniworld. An update to the database intended to reflect a certain change in the miniworld is indeed interpreted as that same change by every observer of the database. However, since a DBMS cannot divine the true state of the miniworld, it would be too much to ask for preservation of meaning in this ideal sense. Instead, one settles for lesser guarantees. *Static consistency* means that the current state of the database is always drawn from a predefined set of consistent database states, which are considered to be valid and unambiguous descriptions of possible states of the miniworld. *Dynamic consistency* ensures that state transitions take consistent states to consistent states. To enforce the two, the appropriate sets of states and state transitions need to be defined in a *database schema.* Usually a database schema can be considered as a database type together with further state constraints. The polymorphic transition operators ensure consistency by observing the schema. Generally speaking, then, consistency refers to the degree of information parity between database and miniworld.

3. *Persistency.* Durability calls for the preservation of data on non-volatile storage, i.e. on a medium with an (at least conceptually) unlimited lifetime. Moreover, preservation should be restricted to those database states that are regarded consistent. Such states are called persistent. As a rule, only the outcome of executing a transactional procedure (called a (database) *transaction*) is considered to be persistent.

4. *Resilience.* A robust DBMS must be able to recover from a variety of failures, including loss of volatile system state, hardware malfunctions and media loss. All failure models to achieve robustness should be based on the notion of consistency. A distinction must be made, though, whether a transaction is affected in isolation or by conflict with others. In the first failure case the failure model causes the database to revert to an earlier persistent state or, if this proves impossible, to somehow reach an alternative consistent state. In the second case, the failure model is one of synchronizing the conflicting transactions so that the result is persistent both from the perspective of the individual transaction (internal consistency) and the totality of the transactions (external consistency).

5. *Performance.* DBMS functionality must scale up to extremely large volumes of data and large numbers of transactions. Scalability will manifest itself in the *response time* of a transaction. System administrators will instead stress *transaction throughput.* Collectively the two are referred to as DBMS performance.

In classical centralized database management systems, *ubiquity* is usually ignored. The topic becomes more important in networked database management systems.

3.2 Physical and Economical Bottlenecks

As noted in Sect. 2.2, utility is the raison d'être for DBMS, but so is durability. Moreover, both are closely intertwined: the former assumes the latter. Consequently, the first step is to analyze whether any of the remaining features directly affects one of them and thus indirectly the other. We claim that the feature that exerts a major influence is performance.

The effect of performance is due to constraints that originate with the physical resources. Even after decades durability is still served almost exclusively by magnetic disk storage. If we use processor speed as the yardstick, a physical bottleneck is one that slows down processing by at least an order of magnitude. 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. This bottleneck is followed by a second one of three orders of magnitude, transmission bandwidth.

We note that even main memory introduces one order of magnitude. Computers attempt to overcome it by staging data in a fast cache store. DBMS should follow a similar strategy of *data staging* by moving data early enough from peripheral to main storage. But now we observe a second, economical bottleneck: The price per bit for main memory is by two to three orders of magnitude higher than for magnetic disk. Data staging that finds a suitable balance between physical and economical bottlenecks will thus have to be one of the principles guiding the architectural design of DBMS.

3.3 Primary Tradeoff: Balancing Data Model and Performance

Section 3.2 suggests that utility and performance should dictate the design strategy, i.e., that they should have first priority. We thus introduce two diametrically opposed design directions: A top-down direction of mapping the data model to the physical resources, and a bottom-up direction of lessening the bottlenecks by data staging (Fig. 3). This is a complicated undertaking that by tradition requires a stepwise approach in order to break up the decision space into smaller portions. The result is a multi-layered architecture.

Determining the decision space for each layer is the paramount challenge. Our conjecture is that we have to find suitable abstractions for both utility and performance such that both match for the purpose of balancing the needs. Utility is in terms of function mapping, and performance in terms of criteria for "early enough" data staging. Figure 4 illustrates the approach. Mapping utility downwards is equivalent to reducing the expressiveness of the data model. Moving performance upwards is equivalent to widening the context for determining what "early enough" means.

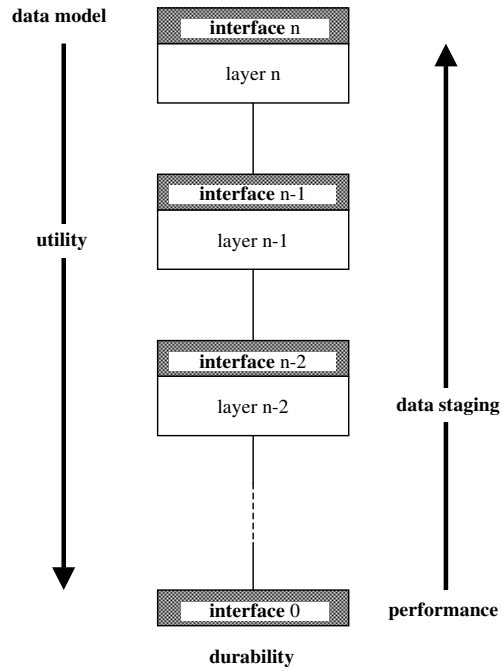

Fig. 3. Architectural design centered on data model and performance

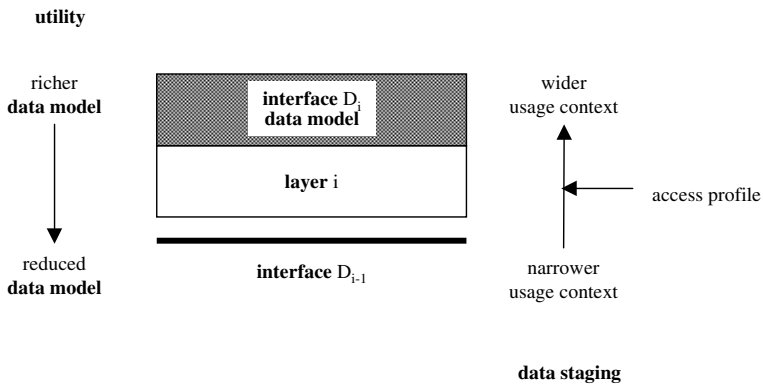

Fig. 4. Balancing functionality and performance within a layer

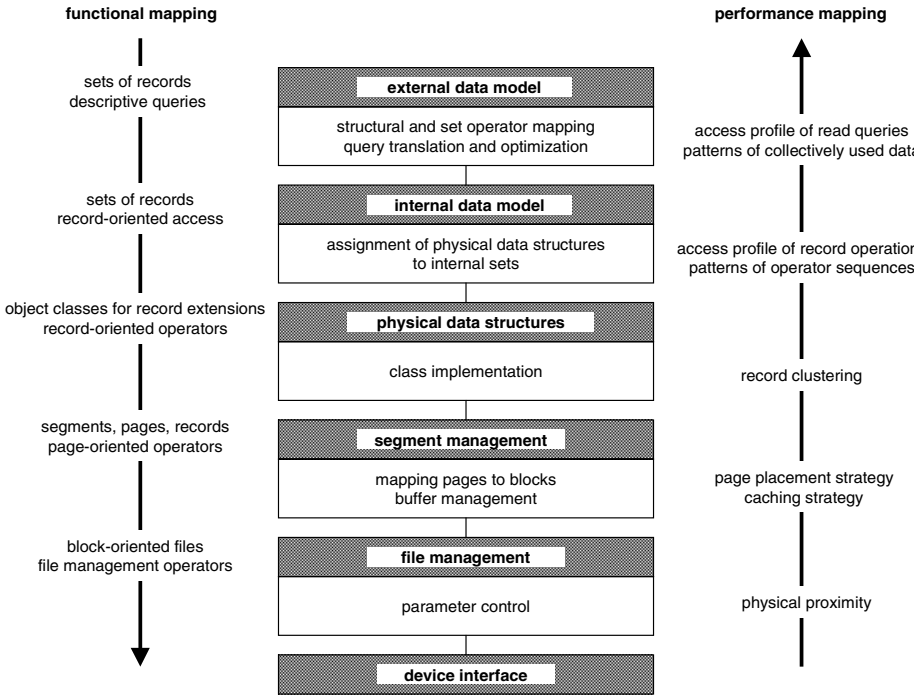

Fig. 5. Reference architecture for set/record-oriented database management systems

A well-known reference architecture for DBMS uses five layers. Figure 5 shows the architecture. We demonstrate that this architecture can be explained in terms of the two factors data model and performance, following the principal ideas just mentioned.

1. We assume a service-level data model that is set/record-structured and uses a set-oriented and, hence, descriptive query language such as SQL or OQL. For data staging on the topmost layer we assume a predominance of read queries. Consequently we examine the sequence of read queries and analyze the result for frequent patterns of collectively retrieved data (read profile). Data would then be staged by rearranging the database according to these patterns. This in turn determines the data model for the next lower layer (internal data model): It should again be set/record-structured.
2. We now turn to the functional mapping. Some reduction in expressiveness should take place. Since structurally there is little difference, the difference can only be in the operators. These will now be record-oriented, i.e., navigational. The mapping encompasses three aspects. One is the actual rearrangement of the external structures. The second concerns the translation of the queries into set-algebraic expressions over the rearranged database, the algebraic and non-algebraic optimization of the expressions with a cost function that minimizes estimated sizes of the intermediate results. The third pro-

vides implementations of the algebraic operators in terms of the next lower data model.

3. We alternate again with data staging. We analyze the entire query profile, now in terms of the record operators. Our hope is to find characteristic patterns of operator sequences for each internal set. Each pattern will determine a suitable data organization together with operator implementations. Consequently, the data model on the next lower layer offers a collection of something akin to classes (physical data structures).
4. Back to the functional mapping we mainly assign physical data structures to the internal sets.
5. At this point we 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 control over physical proximity: adjacent block numbering should be equivalent to minimal latency on sequential access. Unfortunately, not all operating systems offer this degree of control. The data model is defined by classical file management functions.
6. The next upper layer, segment management, is particularly critical from a functional perspective in that it forms a bridge between a world that is devoid of service content and just moves blocks of bytes around, and a world which has to prepare for the services. The bridge maintains the block structure, but makes the additional assumption that blocks contain records of much smaller size. This requires a more sophisticated space management within as well as across blocks and an efficient dynamic placement and addressing scheme for records. Because of all this added value the data model refers to blocks as pages and files as segments. The operators determine access to pages and records and placement of records.
7. The segment management will also determine the success of data staging on the upper layers. For one, if the physical proximity of blocks is to be exploited for pages, mapping of pages to blocks is critical (placement strategy). Even more critical is how the layer exploits main memory to improve page access by three to five orders of magnitude. Use is made of large page buffers. Aside from buffer size the crucial factor is the predictive model of "early enough" page loading (caching strategy).
8. This leaves the details of the physical data structures layer. 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 functional mapping is then concerned with the algorithmic solutions for the class methods.

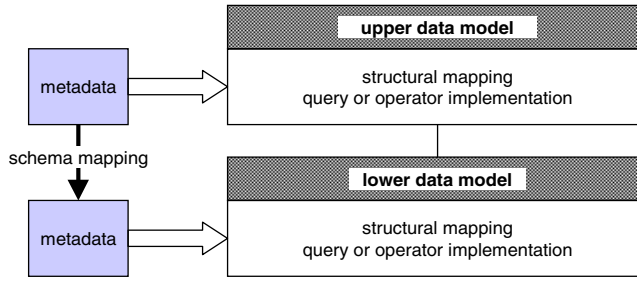

Fig. 6. Principle for adding schema consistency

3.4 Schema Consistency

If data model and performance are indeed the most critical design factors, then the remaining features should only have an orthogonal influence.

Consider the functional mapping in Sect. 3.3. It was entirely based on data models and, hence, generic. One of the features to be added at this point is schema consistency. Schema consistency is equivalent to type safety in programming. In generic solutions type checking is done at runtime. Consequently, type information – often referred to as *metadata* – must be maintained by the operations in each layer. The content of the metadata on each layer is derived from the metadata on the next upper level, by a mapping that is determined by the functional mapping for the data models. On the topmost layer the database schema constitutes the metadata. Figure 6 illustrates the principle, and demonstrates that consistency is indeed orthogonal to the data model.

Since from a data management perspective metadata is just data, the metadata of all layers are often collected into a separate repository called the *data dictionary*, thus perfecting the orthogonality (Fig. 7).

3.5 Consistency, Persistency and Resilience

Transactions define achievable consistency and persistency. They also incorporate the failure model for resilience – atomicity for failures and isolation for suppression of conflicts. Such transactions are said to have the ACID properties. Transaction management consists of three components: a transaction coordinator that does all the interaction with the clients and the necessary bookkeeping, a scheduler that orders the operations of a set of concurrent transactions to ensure serializability and recoverability, and a recovery manager that guarantees persistency and resilience.

Since buffer managers are in charge of a critical part of performance they operate fairly autonomously. On the other hand, atomicity requires close collaboration between recovery manager (including a log manager) and buffer manager. Therefore, recovery managers are made an integral part of segment management. If we wish to achieve orthogonality we should concentrate most or all other tasks of transaction management within this layer, i.e., within segment management.

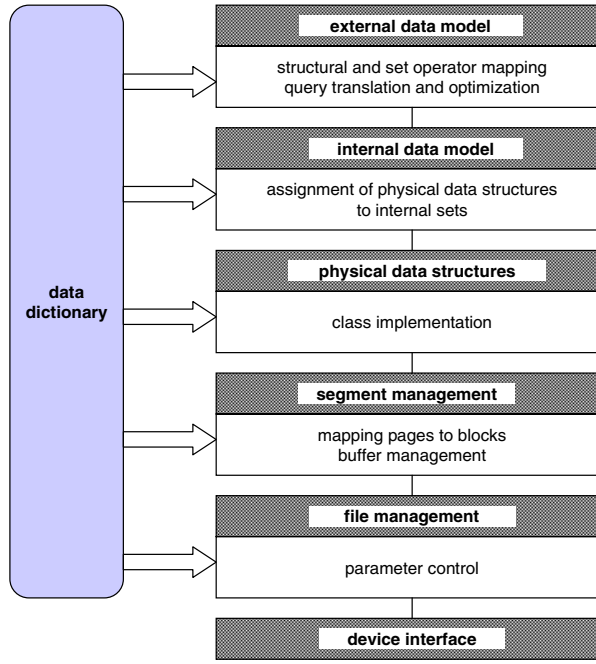

Fig. 7. Data dictionary as an orthogonal system component

As a consequence, we make schedulers a part of segment management as well, and synchronize on the basis of page accesses. Transaction coordinators remain outside the basic architecture because they are uncritical for performance. Figure 8 illustrates the solution.

4 Semistructured Database Management Systems

4.1 Service Features

In Chap. 3 we identified the data model as one of the predominant features. If we continue to verify our refined hypothesis, it makes sense to modify the assumptions for the data model. In this section we replace the data model of sets of small-size records and set operations by a data model for semistructured databases. The model reflects attempts to take a database approach to the more or less structured document databases of the Web or classical information retrieval.

1. *Data model.* The record of the reference model is replaced by a graph structure. A node is labeled by a non-unique tag, and even siblings may have identical tags. A node may include text of any length and optionally a traditional attribute/value list. From the system perspective the text is atomic.

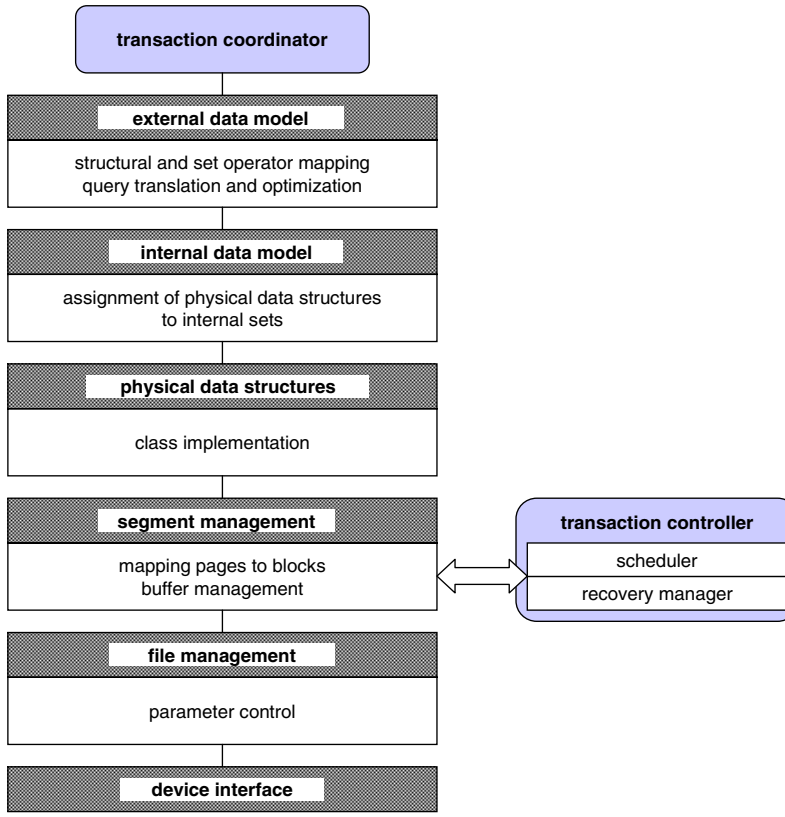

Fig. 8. Incorporating transaction management

Retrieval is considered the predominant service function and is, for all or parts of the graph structure and even across sets of graphs, by pattern matching. Changes to a graph structure, no matter how complicated, are treated as atomic.

The choice of data model has consequences for some of the other features.

2. *Consistency.* Regularity of structured databases is the basis for schema consistency. Whether the DBMS can guarantee information parity even in the limited form of schema consistency depends on the degree of regularity that one may observe for a semistructured database. Regularity will have to be defined in the less restrictive form of regular expressions. Graphs added to the database must then conform to the schema. Because consistency is defined on single graphs and updates to them are already atomic, transactional procedures seem to play a lesser role.
3. *Persistency.* With lesser need for transactions, durability usually implies persistency.
4. *Resilience.* Without transactions, a different failure model must be employed. Failures may occur as well, but there are no longer any fallback positions

that can automatically be recognized by the DBMS. Instead, clients must now explicitly identify persistent states, so-called checkpoints, to which to revert on failure. Since changes to a graph structure are atomic, the failure model does not have to take interferences into account.

5. *Performance.* The physical bottleneck remains the same. Consequently, data staging will have to remain the primary strategy for performance enhancement.

4.2 Architectures

Low Consistency. The notion of database schema makes sense only in an environment with some degree of regularity in database structures. Consequently, where there is no such regularity there is little to enforce in terms of consistency. There are further drawbacks. Little guidance can be given to the analysis of access profiles, and since there is no notion of type extension, little opportunity exists for descriptive access across sets and thus for query optimization. Hence, the upper two layers in terms of the reference architecture of Chap. 3 can exert only very little influence on performance, and there are no major challenges to functional mapping. We may thus collapse the layers into a single one.

There is a strong notion of physical proximity, though. Proximity is defined by the topology of a graph structure, and because reading access is by pattern matching, the topological information should be clustered on as few file blocks as possible. Since long text fields in the nodes are the biggest obstacle to clustering, these should be separated out and kept on other blocks, again in a clustered fashion. Contrary to the reference architecture of Chap. 3 where, due to small record sizes, the individual page is the primary object of concern, performance now dictates that clusters of blocks are considered.

Two tasks remain, then. For performance, clusters of adjacent blocks of data must be staged in main memory. For utility, the graph topology and node fields must be mapped to block clusters. The tasks become part of a layer that replaces the layers of physical data structures and segment management in the reference architecture. With a transaction concept lacking, no further factors exert an influence. Figure 9 shows the result of the discussion.

Schema Consistency. The solution of the previous section treats semistructured databases in isolation. This is against recent trends of combining the traditional structured databases of Chap. 3 with semistructured databases. Suppose that for reasons of *transparency* we maintain a single data model, and we agree on semistructured data as the more general of the two data models. Clearly though, incorporating structured databases forces us to include consistency as an important design factor. In turn we need graph structures that are similar in terms of the underlying regular expressions. These will have to be classified into types. Consequently, even though one will observe less regularity across their instances as in the traditional case there is a notion of database schema possible. To gain set orientation, access should cover complete type extensions so that

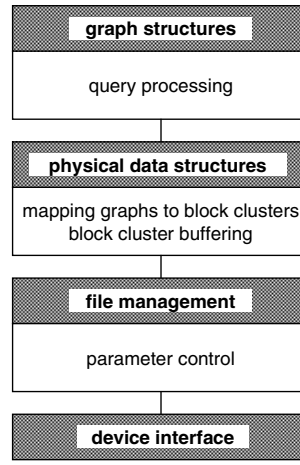

Fig. 9. Architecture for low-consistency semistructured database systems

query languages may again be descriptive, though their concern is retrieval only. Analyses of retrieval profiles may be done along the lines of the graph types. These are all arguments that favor an upper layer similar to the one in the reference architecture of Chap. 3. In the details of functional mapping this layer will in all probability be much more powerful.

From a performance standpoint the retrieval patterns should have an influence. Our conjecture is that a notion of proximity can again be associated with these patterns. If we assume that reading access is limited to extensions of single type, proximity can be translated to subgraphs that correspond to query patterns. Performance mapping, then, consists of dividing the original graph into (possibly overlapping) subgraphs. As above, the graphs should be separated into topology and long text fields or, more generally, arbitrary media data. Functional mapping now includes query optimization.

Subgraphs and long fields require different underlying implementations. In fact, by closer scrutiny of the subgraphs we may detect that some exhibit the strong regularity of traditional structured databases. Hence, there is a need for assigning different physical data structures and, consequently, a need for the second layer of our reference architecture. However, whereas the uppermost layer seems to be more complicated than the one in Chap. 3, the reverse seems to be true for the second layer.

For the implementation we take a two-track approach. There is a good chance that subgraphs have a size that can be limited to single blocks. Hence, their implementation may follow the reference architecture of Chap. 3 from the physical data structures layer on downwards, though in detail the techniques may differ. Fields with unstructured media data should follow the architecture of Fig. 9. Figure 10 summarizes the discussion.

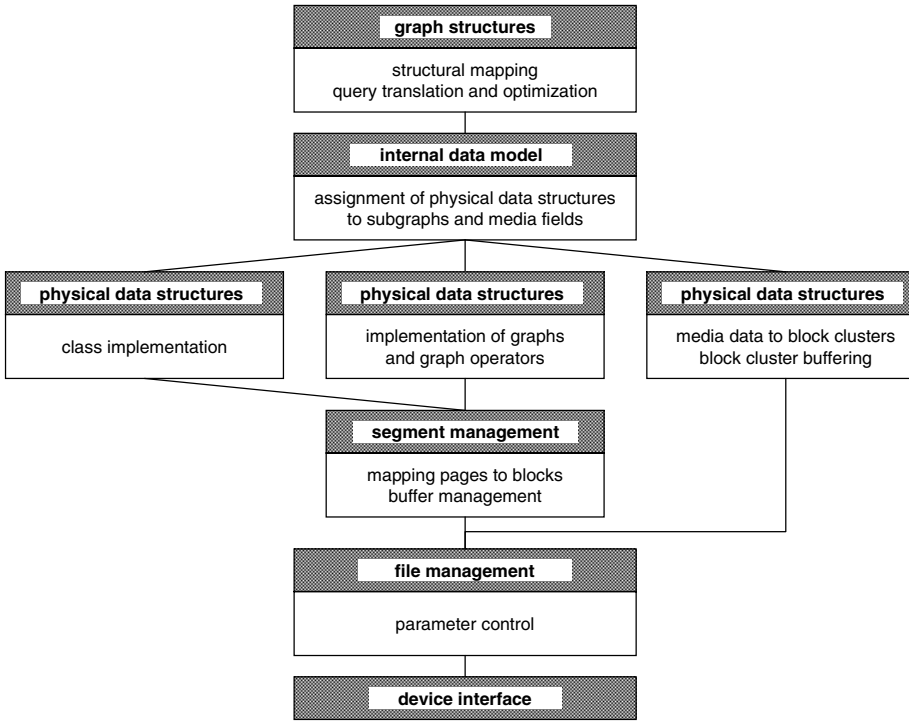

Fig. 10. Architecture for schema-consistency semistructured database systems

5 Multimedia Databases

5.1 Service Features

Even though one may attach any type of media data to a graph node, and the types may vary, semistructured databases are by no means multimedia databases. They become multimedia databases only if we offer services that take note of the interrelationships between different media and, hence, are capable of combining two or more media. The new service functionality, *cross synchronicity*, ensures that the contents of the related media match along a time scale. Take as an example the continuous playback of combined video and audio.

Cross synchronicity relies on *playback synchronicity*, the ability of a system to present media data under temporal conditions identical to those during recording. As a service feature, playback synchronicity straddles the line between functionality and quality. Poor playback synchronicity limits utility, but there is a certain tolerance as to speed, resolution and continuity of playback within which utility remains preserved. Hence, playback synchronicity could also be regarded as a service quality.

5.2 Architecture

To study the architectural effects we presume that cross synchronicity is a service functionality and playback synchronicity a service quality. Since cross synchronicity is a feature over and above those that gave rise to our reference architecture, we should try to treat it as an orthogonal feature, and encapsulate it in a separate layer or component somewhere on the upper levels.

Playback synchronicity seems closely related to performance, because it has to deal with the same kind of bottleneck – peripheral storage. However, conditions are even more stringent. Playback must remain fairly continuous over minutes up to an hour or so. Longer breaks or jitter due to relocation of access mechanisms or contention by other processes must stay within guaranteed limits. Hence, depending on speed special storage devices (stream devices) or dedicated disk storage are employed. Special buffering techniques will have to even out the remaining breaks and jitter. The layers further up should intervene as little as possible. Hence, for continuous media the right-hand branch in Fig. 10 must become even more specialized. The physical data structures layer should now restrict itself to storing an incoming stream of media data and managing it. Discontinuous media data and graph structures could follow the architecture of Fig. 10.

Now, given playback synchronicity of each individual media data, the upper layers of Fig. 10 may be preserved to deal with the structural aspects of the interrelationships between the media, again intervening only during storing of the data and during the setup phase of playback. On top we add another layer that controls cross synchronicity, usually along the lines of a given script. Figure 11 gives an overall impression of the architecture.

6 Networked Databases

6.1 Service Features

We return to Fig. 1. In distributed information systems the single business process and even a work unit faces a multiplicity of resource managers. All these may be placed at geographically disparate locations. Consequently, *ubiquity* enters the picture as an additional service functionality.

In Sects. 2 and 3.1 the service functionalities of consistency and persistency and the service quality of resilience where tied to the notion of database transaction and, hence, to a single resource manager. In general, these features ought to be tied to the business process as a whole. Technically then, we refer to a business process as an *application transaction*. Usually, an application transaction is distributed. Since each component deals with transactional properties on an individual basis, *coordination* comes into play as a further service functionality.

Service clients as well as service providers differ in a large number of characteristics. Some of these are entirely technical in nature, such as differences of hardware and operating system platforms or the transmission protocols they understand. These discrepancies are referred to as technical heterogeneity. Other

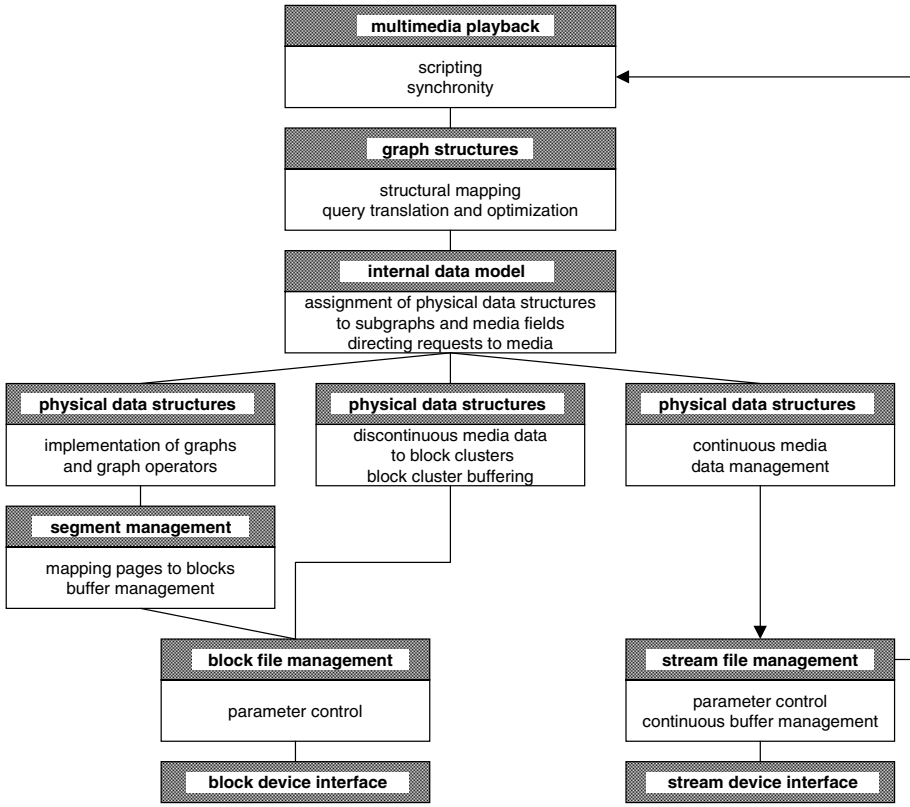

Fig. 11. Architecture for multimedia database systems

differences – noted as semantic heterogeneity – have to do with functionality, e.g., different data models, or consistency, e.g., different schemas. How much heterogeneity is acceptable seems more of a gradual decision. Consequently, we add a new service quality: *homogeneity of services*.

6.2 Middleware

If we wish to prove our hypothesis, we should follow our previous architectural strategy and realize the new features orthogonally whenever possible. In the case of distributed information systems, the strategy translates into touching the resource managers only lightly, and adding infrastructure that deals with the new features. This infrastructure goes under the name of *middleware*.

The first issue to deal with is ubiquity. The issue is resolved by utilizing the data communications infrastructure and employing a common high-level protocol, e.g. TCP/IP. The second is technical homogeneity. Middleware enforces the homogeneity by establishing an internal standard (take the IIOP protocol of CORBA) and requiring site-local adapters. Practically all modern middle-

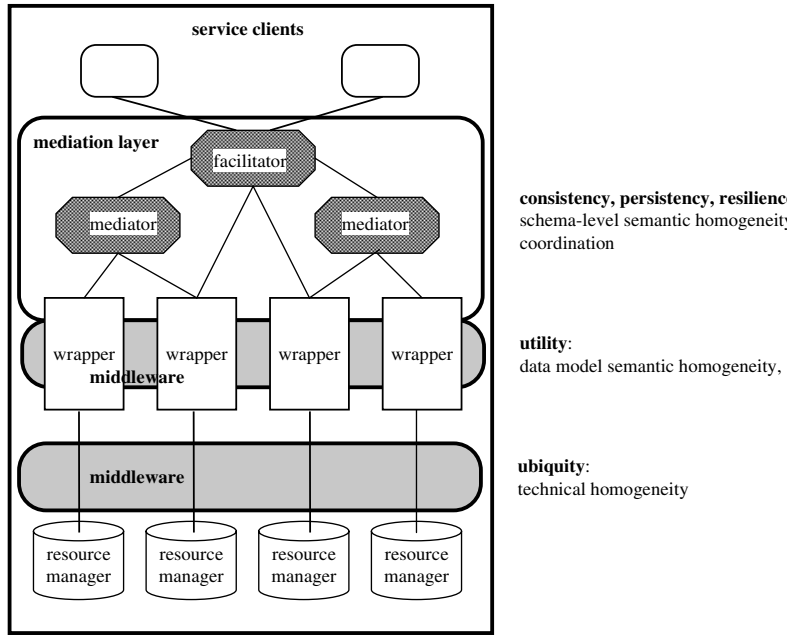

Fig. 12. Reference architecture for middleware

ware approach semantic homogeneity by setting their own data model standard (e.g., HTML documents for the WWW, remote objects for CORBA, DCOM and RMI).

This leaves as tasks those that are much more difficult to standardize, essentially all those that have to do with content. Content affects consistency and derived features such as persistency or resilience. If one cannot come up with standards, meta-standards may help. The architectural representation of meta-standards is by frameworks. A framework that is oriented towards semantic homogeneity on the schema level, and coordination issues, is the I^3 (Intelligent Integration of Information) architecture. Figure 12 combines the middleware and I^3 approaches into a single architecture.

Wrappers, mediators and facilitators are part of the I^3 framework. Wrappers adapt their resource managers on the semantic level, by mapping the local data model to the common, standardized data model and the schema to one expressed in terms of the common data model. Mediators homogenize the schemas by overcoming discrepancies in terminology and structure. Increasingly they rely on ontologies that are represented by thesauri, dictionaries, catalogues, or knowledge bases. Other mediators accept queries that span several resources, translate them according to the homogenized schemas, send them off to the resource managers, and collect the homogenized results. Facilitators ease the orientation of service clients in the network. Examples of facilitators are Web search engines or catalogues of data sources.

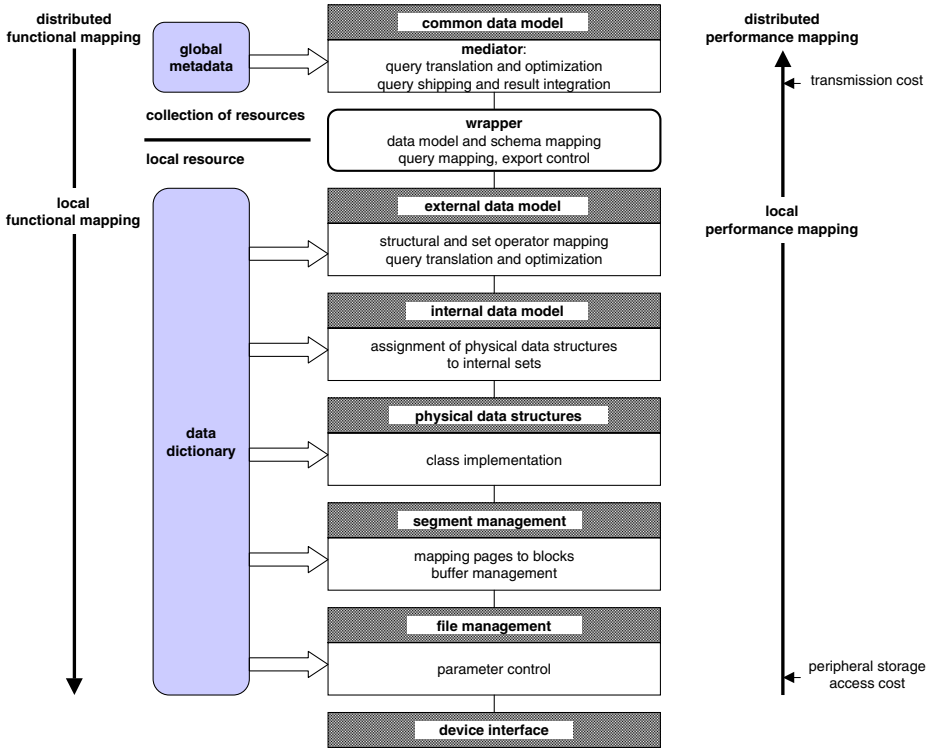

Fig. 13. Database management system in a network

6.3 Data Model, Consistency and Performance

Local utility and consistency are the responsibility of the individual resource manager. Collective utility and consistency are the realm of the mediator. From the viewpoint of the individual database management system all aspects of global utility are an add-on service functionality and, hence, relegated to a new top layer. Figure 13 illustrates this logical view. The view leaves open whether the mediator is a centralized component or distributed across the resource managers.

Mediators achieve on a network-wide basis what the external data model layer does locally: query translation, query optimization, query shipping and result collection and integration. These are tremendously complicated issues, but nonetheless orthogonal to all the local tasks. The mediator is supported by metadata structures such as the catalogues and ontologies mentioned before, or a global schema (the so-called federated schema).

The wrapper, besides mapping the schema, queries and results to the common data model, acts as a kind of filter in order to make visible only those parts of the database which are globally accessible (the so-called export schema).

Because the local database system remains unaffected, all new performance bottlenecks arise from distribution. In the past it was transmission time that

dominated the mediation strategies, and it remains so still today to the extent that bandwidths have not kept pace with processor speed. Independent of speed, latency, i.e. the time delay between request and start of transmission, remains a serious handicap. Therefore, query optimization is largely governed by transmission cost.

6.4 Consistency, Persistency and Resilience

Globally, the service functionalities of consistency and persistency and the service qualities of concurrency and resilience are tied to an application transaction. Locally, the functionalities and qualities are guaranteed by ACID database transactions. We limit our considerations to application transactions that also are ACID. Consequently, unless it suffices for global isolation and atomicity to rely solely on local isolation and atomicity, some global – possibly distributed – coordination mechanisms must be introduced.

It is well known from transaction theory that local isolation is the basis for any kind of global isolation. Consequently, conflicts among business processes (see Fig. 1) are handled purely locally. Likewise it is well known that global atomicity – global persistency and resilience – require coordination in the form of global commit protocols. These protocols are centralized in the sense that they distinguish between a coordinator and its agents, where the agents reflect those resource managers that updated their databases. The most widely used protocol, Two-Phase-Commit (2PC), requires that even after local commit the agents are capable, during the so-called uncertainty phase, of rolling back the transaction. In purely local transaction management an uncertainty phase does not exist. As a consequence, the recovery manager must suitably be adapted.

This leaves the question of where to place coordination and, incidentally, transaction monitoring. Both are generic tasks. If we assume that application transactions pass through the mediator, it seems only natural to place the tasks with the middleware (Fig. 14). And indeed, this is what middleware such as CORBA and DCOM try to do. In fact, today's understanding of many of the earlier transaction processing systems is that of middleware.

In summary, accounting for distributed transaction management is still close to orthogonal: A global component is added, and a single local component is adapted.

7 Conclusions

Does the paper support our hypothesis that architectural design plays *the* strategic role in identifying, articulating, and then reconciling the desirable features with the unavoidable constraints under which a system must be developed and will operate? Is the design philosophy and methodology with a first phase for identifying the desirable features and the constraints and a second phase for determining the tradeoffs the correct consequence? We claim the paper does.

Equally important – and this seems the novel aspect of our approach and thus the contribution of the paper – is a refinement of the strategy that challenges

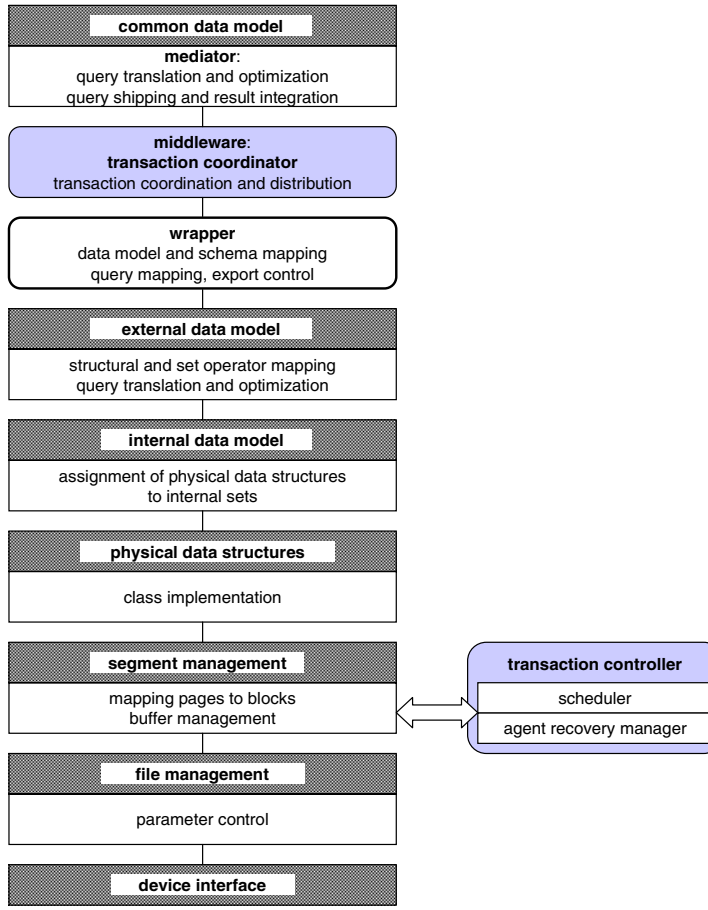

Fig. 14. Incorporating global transaction management

designers to identify – very much in the abstract – those features that promise to dominate the design of the architecture. If successful, the remaining features in the second class can be treated orthogonally. Orthogonality may either be perfect so that the features give rise to additional components, or at least sufficiently strong so that the features can be taken care in a single component within the architecture developed so far.

The proof was by circumstance and limited to one kind of system – database management systems in networks. Architectures that have proven their worth in the past were evaluated in retrospective. Others – DBMS architectures for semistructured and multimedia databases – still vary widely, and the ones we developed in this paper reflect those in the literature that sounded most convincing to the author. Also, there is by no means universal agreement on how to divide the responsibilities for the service features between resource managers

and middleware. Our hope is that this paper will contribute to future design decisions.

One may argue that our base was way too small and too specialized to render statistically significant evidence. Clearly, the study should continue to cover other kinds of systems. Our next candidate for attention is data communications systems. Nonetheless, it should have become clear that a design strategy based on our hypothesis is little more than a conceptual framework. Architectural design of software systems remains a creative task, albeit one that should follow sound principles of engineering.

8 Bibliographic Notes

The reference architecture of Chap. 3 is due to T. Härder and A. Reuter and is itself based on the early System R architecture. A modern interpretation can be found in Härder, T.; Rahm, E.: *Datenbanksysteme: Konzepte und Techniken der Implementierung.* Springer, 1999 (in German). Many of the numerous and excellent textbooks on database systems that have appeared in the more recent past use similar architectures as a reference. Where publications on commercial database products present system architectures (unfortunately not too many do) they seem to indicate that overall the same principles were applied. A careful analysis of peripheral storage as the performance bottleneck is given in Gray, J., Graefe, G.: *The Five-Minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb.* ACM SIGMOD RECORD, 1998. Among the textbooks and publications on semistructured database systems hardly any deal with issues of architecture to any detail. Of those on multimedia databases the situation is only slightly better. Architectures can be found in Apers, P.M.G.; Blanken, H.M.; Houtsma, M.A.W. (eds.): *Multimedia Databases in Perspective.* Springer, 1997 and Chung, S.M. (ed.): *Multimedia Information Storage and Management.* Kluwer Academic Publ., 1996.

The classical textbook on transaction management, which starts from an architectural view is Gray, J., Reuter, A.: *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publ., 1993. An architectural approach is also taken by Bernstein, P.A., Newcomer, E.: *Principles of Transaction Processing.* Morgan Kaufmann, 1997. For middleware the reader is referred to the numerous literature over the past few years. A good pointer to the I^3 architecture is Wiederhold, G.: *Intelligent Integration of Information.* Kluwer Academic Publ., 1996, whereas the details have remained in draft form: Arens, Y. et al.: *Reference Architecture for the Intelligent Integration of Information.* Version 2, ARPA Tech. Report.

Acknowledgements

The author is grateful to Gerd Hillebrand for his thoughtful comments and discussion on an earlier version of the paper. Klaus Dittrich through his comments helped sharpen the focus of the paper.