1 Component Database Systems: Introduction, Foundations, and Overview

Andreas Geppert
Klaus R. Dittrich
University of Zurich
1.1 Introduction

Database management systems (DBMSs) support individual applications and comprehensive information systems with modeling and long-term reliable data storage capabilities as well as with retrieval and manipulation facilities for persistent data by multiple concurrent users or transactions. The concept of data model (most notably the relational models, Codd 1970; and the object-oriented data models, Atkinson et al. 1989; Cattell & Barry 1997), the Structured Query Language (SQL, Melton & Simon 1996), and the concept of transaction (Gray & Reuter 1992) are crucial ingredients of successful data management in current enterprises. Nowadays DBMSs are well established and are, indeed, the cornerstones of virtually every enterprise.

Traditionally, data elements stored in databases have been simply structured (e.g., employee records, and product and stock information). Transactions have been of short duration and often needed to access only a few data items. Most traditional queries have been simple and the techniques used to answer them efficiently are well understood. Taking a broader view, DBMS-based information systems have been built in a rather database-centric way; that is, environment decisions such as the use of mainframe-based or client/server architectures have been typically based on what the DBMS itself requires or supports in this respect.

In the recent past, however, more and more new and demanding application domains have emerged that could also benefit from database technology, and new requirements have been posed to DBMSs. Many applications require the management of data types that are not handled well by conventional DBMSs. Examples of such new data types include multimedia data, documents, engineering artifacts, and temporal data, to name just a few.

Likewise, DBMSs are more and more often required to integrate with other infrastructure parts of their environment. For instance, instead of letting the DBMSs manage already existing data, it is often necessary to leave data where it is (possibly because there are applications that users would not want to migrate as well) and instead enhance the external data management system with some sort of database functionality (Vaskevitch 1994). It might also be necessary to integrate existing data stores with database systems in such a way that each of them is still independently operational, while users are provided with an integrated and uniform view of the entire system. In other words, often applications need support as offered by multidatabase systems or federated database systems (Elmagarmid et al. 1999; Sheth & Larson 1990), in which the federated parties might be any kind of data store. Some applications even require support for queries combining data from databases and data extracted from sources such as the World Wide Web (WWW).
In order to meet all these new requirements, DBMSs (or whatever the resulting kind of system will ultimately be called) apparently have to be extended to include new functionality. However, enhancing a single system with modules implementing all the new functions is not a viable approach for several reasons:

- DBMSs would become so large and, in consequence, complex that they could no longer be maintained at reasonable cost.
- Users would have to pay a high price for the added functionality, even if they do not need every part of it.
- Users and applications would also have to pay a performance penalty for added functionality that they actually do not need.
- A DBMS vendor might not have the expertise to perform such extensions and might not have the resources to undertake all extensions within a reasonable period of time.

Thus, beefing up a monolithic DBMS by adding more and more functions does not work. Instead, it seems attractive to consider the alternative of extending DBMSs in a piecemeal way, that is, striving to allow functionality to be added or replaced in a modular manner, as needed.

Modular extensions to a DBMS require that a well-defined software architecture is imposed on the DBMS. This architecture clearly defines the places in the system where extensions are possible. In general, extensions should be possible at a few well-defined, distinct places in the system only, and the effects of modifications on other parts should be avoided or at least minimized. To that end, DBMS architectures need to be componentized in such a way that new components can be added to it or existing components can be exchanged in a flexible yet well-defined way. Thus, the componentized architecture specifies and restricts the ways in which a DBMS can be customized. Ultimately, the componentized architecture also defines the notion of component, itself, and hence the realm of valid extensions to a DBMS.

We refer to DBMSs that have a componentized architecture and allow users to add components as component DBMSs (CDBMSs). Due to the componentized architecture, these extensions are possible without requiring other system parts to be rewritten. Components can be provided by third parties and possibly even users, thus increasing the developer base of a DBMS. Ultimately, unnecessary components do not have to be added, and applications therefore do not pay (in terms of system cost and performance) for functionality they do not use.

In the remainder of this chapter, we introduce principles and different forms of CDBMSs. In Section 1.2, we present a more detailed motivation of componentization of DBMSs. We then discuss the foundations of CDBMSs: DBMS architecture and componentware. In Section 1.4, we review past efforts regarding the extensibility of DBMSs. Section 1.5 presents a more detailed taxonomy of CDBMSs, and Section 1.6 concludes the chapter.
1.2 **The Need for Componentized DBMSs**

In this section, we will motivate CDBMS in more detail (see also Vaskevitch 1994, 1995). To this end, we consider DBMS support for advanced information systems (IS) with respect to the following tasks: management of new data types, adapted and new database functionality, and better integration with other IS parts and the IS environment.

1.2.1 **Handling Nonstandard Data Types**

In view of the success of relational database systems in supporting IS, new applications pose new requirements for database systems, and existing application areas define extended requirements. For example, many organizations offering services or products in the WWW want to manage their data in a database system. In particular, e-commerce applications need database support to manage their catalogues, product descriptions, orders, and so forth. Other areas, such as engineering (e.g., bio-engineering, mechanical engineering, or software engineering) and scientific applications (e.g., in biochemistry, geography, or meteorology), also extend the requirements of database systems. In those cases where data do not already live in existing systems, it will often be desirable to exploit database technology in order to benefit from data independence, query capabilities, transaction management, and consistency enforcement.

These applications often need support for modeling and storing data that, so far, are not supported by database systems. Examples of such nonstandard data types are images, text, spatial data, temporal data, videos, Virtual Reality Modeling Language (VRML) objects, and structured documents. Moreover, applications often need to aggregate related instances of these data types into complex objects. While some advanced data types might be supported by specialized DBMSs (e.g., temporal, spatial, or multimedia database systems), in many cases a system that offers exactly the desired set of data types does not exist. Moreover, many of these specialized DBMSs are not available commercially.

In addition to storing nonstandard data, the data also need to be retrieved. Thus, declarative queries over these nonstandard data types must be supported. However, many of them have type-specific query modes, such as content-based search in the case of images or information-retrieval-style queries in the case of text documents. Even existing applications (such as financial applications) might need more powerful query support (e.g., new forms of aggregates, such as a moving average).

Query processing, of course, must not only be possible, but it also needs to be efficient. To that end, query optimizers must be able to generate efficient plans for operations on any kind of data supported by the DBMS, including nonstandard data. Optimizers thus must know
(or be taught) how to generate efficient execution plans for queries over predefined as well as nonstandard data types. Furthermore, nonstandard data types often require or can at least benefit from specialized index structures. Without exploiting these index structures, the performance of retrieval is, in many cases, suboptimal. DBMSs therefore should be able to incorporate new index structures whenever these are needed to execute queries against nonstandard data types efficiently.

Currently, no single, off-the-shelf DBMS exists that meets all these requirements for a complete range of data types. Apparently, vendors are not interested in or able to build such a system for several reasons. Such a one-size-fits-all solution might meet the functional requirements of most applications, but for each application it also would provide functionality that this application does not need. Users would still have to pay for all the functions, regardless of what they effectively need. Moreover, adding functions usually leads to performance degradation, even if the new functions are not used—thus, customers have to pay a performance penalty for features they do not use.

A further reason for vendors’ reluctance to build such a system is that adding too many features in parallel is not advisable and manageable. In special cases, a vendor also might not have the expertise to add the required new features, for example, if the implementation of new query modes or optimizations requires highly specialized, type-specific knowledge.

Thus, a single DBMS supporting all the emerging and ever-more-demanding applications is not conceivable. A possible solution is to extend a DBMS on demand—that is, to provide standard, commonly needed functionality right away and to add other, more advanced features in a piecemeal way later. Customers can then purchase the core DBMS plus the extensions they require. In other words, instead of a monolithic DBMS, customers use an a la carte DBMS configuration.

### 1.2.2 Data Integration

The analysis just described considers cases where all the data are kept, or can be brought, under the control of a single DBMS. This, however, is not always possible because in many cases the data have existed in specialized data stores, such as image libraries or document collections, for a long time. The migration of data and application programs without loss is often difficult or impossible. Hence, existing and new data are to be integrated under the common roof of an integration layer. The reasons for integration typically include the facts that data in different data stores are (semantically) related in some way and that the task of the integration layer is to store and maintain these relationships. Likewise, integration is often desired to achieve a uniform and homogeneous view of the diverse data stores and to receive database support.
for the integrated system (e.g., queries against the integrated system). Thus, we face the challenge of integrating a broad diversity of data stores into a database environment.

In order to integrate disparate data stores, some kind of middleware system is needed that understands (or can be taught) the data models of the data stores. Even in the case of standardized data models (such as the relational one), each system has intricacies that require a system-specific handling and mapping. Even worse, nondatabase systems handle data in an application-specific format. Consequently, integration cannot be done just once per system, but has to be done separately for each data source.

As in the previous case, there are two options: Either a DBMS is built that knows everything (i.e., how to integrate all the possible data stores) or the integration platform is kept extensible and built in a piecemeal way. The first option is, again, not feasible because the DBMS has to know about all the interfaces and intricacies of the involved data stores.

A feasible solution to this problem is, once more, to use the principles of componentization and extensibility. Assume the middleware layer defines a (minimal) set of functions it expects from the data stores. Each extension then implements these functions using the interface of the corresponding data store to be integrated. In other words, these extensions serve as gateways or wrappers: They abstract from the intricacies of the data store, and they (instead of the data stores proper) are hooked into the middleware. The apparent advantages of this approach are that the knowledge and customization required of the middleware layer are minimized (because it only needs to know the general abstract notion of wrappers) and that ideally wrappers or gateways are written once per type of data store.

### 1.2.3 Downsized Database Systems

So far, we have implicitly assumed that required database functionality is provided in the form of database systems, and that nonstandard requirements are met by adding extensions. In order to use database functions, applications are implemented on top of a database system or database middleware. This, however, is not the only conceivable form of database support to meet novel requirements. Sometimes, applications need only parts of the entire spectrum of database functions or need these parts in a very specialized form. Forcing applications to always use full-fledged, entire DBMSs can turn out to be an overkill and/or to not lead to the right kind of support. As an example, suppose a document management system or a groupware system already has its own persistence mechanism to store documents. Assume further that queries are desired, but not yet supported. Query support could be achieved by moving the document management system on top of a
1.2 The Need for Componentized DBMSs

Of course, splitting a DBMS into single, stand-alone parts is a difficult problem because the right balance between the granularity of services and the (number of) interservice relationships has to be found. Likewise, determining the right interfaces for the services is a complex task. However, feasible solutions to these questions and, in consequence, the availability of a broad range of services open up completely new ways of implementing database applications and application servers. These systems can then pick the service provider that is optimal for their problem and do not need to rely on large full-fledged DBMSs. Moreover, in cases in which services are standardized, we can envision that in such a scenario it will also be possible to mix and match services from different vendors. The objective of turning DBMSs into lightweight collections of freely combinable services can be met only if these services and their implementations are described and built as components.

1.2.4 Discussion

The general problem in the scenarios we considered is the monolithic structure of a traditional DBMS. By monolithic structure or monolithic architecture, we mean that the DBMS is a single unit whose parts are connected to one another and dependent on one another to such a degree that modifications and extensions are not easily possible. In particular, each DBMS part might make assumptions about the requirements and operations of other parts, which leads to domino effects whenever changes are made in one place. Extensions can only be made if all the interconnections and dependencies are known. The evolution of and extensions to a DBMS are only possible by the vendor, who, for each extension, also needs to make sure that other affected parts are adequately adapted.

To prevent misinterpretations, it should be stressed that monolithic architectures in their current common form have not emerged simply because developers and vendors did not know better. In the past, they have been sufficient because applications posed rather restricted requirements for DBMSs. Moreover, a monolithic DBMS can be implemented in a way that optimizes runtime performance and throughput for all applications that need just the functionality offered by the DBMS. Nevertheless, whenever extensions and customizations are considered, problems with monolithic architectures occur with respect to system complexity, system performance, system cost (production and maintenance), and complexity of system evolution.

The general idea for overcoming these problems and still providing for the needed functionality as sketched in the previous scenarios is to
offer a core system and to extend it as needed. CDBMSs are meant to allow such extensions in a controlled and safe way. Although different forms of CDBMSs exist, their common basis is a componentized architecture and the support of components implementing some kind of database function. These components can be added to a DBMS or used in some other way to obtain database support.

In order to effectively allow extensions, the DBMS architecture must be made explicit and be well defined. While typically some parts of the CDBMS will need to be fixed without the possibility of altering them, others can be extended (we call this the variable part). Explicit means that the system structure is defined, preferably in a (formal) architecture model, and that the system structure is visible to those actors modifying it. Second, for a CDBMS the meaning of the notion of component is defined, with varieties ranging from abstract data types to implementations of internal tasks. However, the common characteristics of components are that they implement a coherent set of functions, make all restrictions concerning their use explicit in their interface, and are generally applicable across a variety of applications. Ultimately, a CDBMS architecture also defines places in the system (the variable part) where components can be added. These places can be thought of as hooks used to plug components into the enclosing system. Technically, these hooks are defined in terms of the interfaces the component can use and/or should implement.

1.3 Prerequisites and Foundations of CDBMSs

In this section, we elaborate on the principles of CDBMSs in more detail. As mentioned before, extensions to a DBMS affect its architecture and also require certain prerequisites to be met. We therefore first briefly address the issue of DBMS architecture. The subsequent section relates CDBMSs to componentware and then gives a classification of CDBMSs.

1.3.1 DBMS Architecture

Different kinds of architecture serve different purposes. For instance, the three-level-schema architecture (which distinguishes the external schemas that users work with, the internal integrated schema of the entire database, and the physical schema determining the storage and organization of databases on secondary storage) reflects the different levels of abstraction of data in a database system. The layered architecture in Figure 1.1 illustrates a hierarchy of mappings, where the topmost layer deals with data model entities and the bottommost layer deals with blocks and files. Finally, a task-oriented architecture identi-
fies the relevant modules (i.e., their purpose and interface) and relationships to other modules (e.g., in the form of exported and imported interfaces). Examples of such tasks include query optimization, concurrency control, and recovery. The last two also are examples of tasks that are hard to assign to a specific layer in the layered architecture or that might even be addressed by multiple layers. Although a task-oriented architecture is much more suitable for reasoning about extensibility and DBMS construction, reference architectures rarely exist (with the strawman architecture developed by the Computer Corporation of America, CCA 1982, as a notable exception), and concrete architectures are described at a granularity too coarse to be helpful for our purposes.

For educational purposes, it is convenient to consider a DBMS architecture as consisting of a number of layers (Härder & Rahm 1999; Härder & Reuter 1983; Ramakrishnan 1997). Each layer supports a set of data types and operations at its interface and typically consists of several components (modules or managers of some concrete or abstract
The data types and operations defined for the modules of one layer are implemented using the concepts (data types and operations) of the next-lower level. Therefore, the layered architecture can also be considered as a stack of abstract machines. Concretely, the layered architecture model as introduced by Härdner and Reuter (1983) is composed of five layers (see Figure 1.1):

1. The uppermost layer (L4) supports logical data structures such as relations, tuples, and views. Typical tasks of this layer include query processing and optimization, access control, and integrity enforcement.

2. The next layer (L3) implements a record-oriented interface. Typical entities are records and sets (e.g., as found in the Committee on Data Systems Languages, CODASYL data model) as well as logical access paths. Typical components are the data dictionary, transaction management, and cursor management.

3. The middle layer (L2) manages storage structures (internal records), physical access paths, locking, logging, and recovery. Therefore, relevant modules include the record manager, physical access path managers (e.g., a hash table manager), and modules for lock management, logging, and recovery.

4. The next layer (L1) implements (page) buffer management and implements the page replacement strategy. Typical entities are pages and segments.

5. The lowest layer (L0) implements the management of secondary storage (i.e., maps segments and pages to blocks and files).

In general, due to performance considerations, no concrete DBMS has fully obeyed the layered architecture. Note further that different layered architectures and different numbers of layers are proposed, depending on the desired interfaces at the top layer. If, for instance, only a set-oriented interface is needed, it is useful to merge the upper two layers.

From a more practical point of view, most DBMS architectures have been influenced by System R (Astrahan et al. 1976), which consists of two layers: the relational data system (RDS), providing for the relational data interface (RDI); and the relational storage system (RSS), supporting the relational storage interface (RSI). While RDS implements SQL (including query optimization, access control, triggers, etc.), RSS supports access to single tuples of base relations at its interface.

1.3.2 Components and Database Management System Architecture

When we strive for reusability, extensibility, openness, and interoperability of database systems, looking at software engineering research and practice yields helpful insights. In particular, componentware (Allen &
Frost 1998; D’Souza & Wills 1999; Griffel 1998; Hamilton 1997; Krieger & Adler 1998; Nierstrasz & Dami 1995; Nierstrasz & Meijler 1998; Orfali et al. 1996; Szyperski 1997) is a recently proposed paradigm to address these issues. This is the notion that software systems are built in a disciplined manner out of building blocks with specific properties, called components. There is currently no widely agreed-on definition of the term component; however, the following characteristics of components can be found in most definitions in the literature. A (software) component, then, is a software artifact modeling and implementing a coherent and well-defined set of functions. It consists of a component interface and a component implementation. Components are black boxes, which means that clients can use them properly without knowing their implementation. Component interface and implementation should be separated such that multiple implementations can exist for one interface and implementations can be exchanged. Defining components as black boxes also means that each component sees only the interfaces of other components; that is, access to the internal operations and structures of other components is not permitted. A component should not have an overly high number of relationships to other components because this might restrict its reuse potential.

Systems are built by putting components together to form new software systems (this principle has been referred to as reuse by composition). Systems constructed by composition can be modified or extended by replacing or adding new components. Component-based systems are expected to facilitate the addition or replacement of components without recompilation (or even without shutting down) the entire system.

In order to put the right components together to obtain complete and adequate systems, a frame (into which components are plugged) and rules governing the composition process are needed. The frame is given by the software architecture (Perry & Wolf 1992; Shaw & Garlan 1996) of the system under construction. Similar software systems are then described by architecture skeletons or generic architectures (Nierstrasz & Meijler 1998) that are successively enhanced and completed by components. Thus, as a prerequisite, the underlying generic architecture needs to be defined in terms of components (acting as placeholders) and connections in such a way that components can later be added in a meaningful and consistent way.

Components usually possess a coarser granularity than objects in object-oriented systems and models. A well-designed component supports a coherent set of tasks (e.g., in one of our scenarios, storing and retrieving textual documents), while objects and classes typically address only a part thereof. Components and objects are, however, not mutually exclusive alternatives; rather, components leverage object orientation to a higher level of abstraction and granularity. In fact, “under the hood” components are often assemblies of objects.
We use the principles of componentware to better understand, abstract, and classify the various approaches to extending and customizing DBMSs. Moreover, the characteristics of componentware (components and architecture) are crucial requirements for systematic and well-defined extensibility and integration. Extensions to a DBMS in this context are represented as components (i.e., they should meet the aforementioned properties of components). Further, DBMS should exhibit a componentized architecture, at least for those parts that are intended to be customizable.

1.4 Related Work: The Roots of CDBMSs

In this section, we review the roots of CDBMSs. In a nutshell, these roots are relational database systems, object-orientation in general and object-oriented DBMS in particular, and extensible database systems.

Extensible database systems (Carey & Haas 1990) all attempt to ease the construction of DBMSs by exploiting some kind of software reusability (Geppert & Dittrich 1994). The proposal is for a general core that can be customized or extended in some way by users, or even used to generate some DBMS parts. Here, we survey these approaches and classify them by their dominant way of extending or constructing DBMSs.

1.4.1 Kernel Systems

Kernel systems offer the common functionality required by all or most DBMS (e.g., physical object management), but typically are not fully functional DBMSs. The upper layers of a DBMS have to be implemented (i.e., programmed) by the DBMS implementor (DBI). The major question in this approach is how powerful the kernel interface is. A low-level interface (e.g., page management or unstructured records) leaves the DBI enough freedom to implement the desired concepts. On the other hand, much implementation work is necessary due to the low level of the kernel. Alternatively, if a kernel supports more powerful concepts, less implementation is required from the DBI, while the kernel will be geared toward specific kinds of systems or constructions.

The Wisconsin Storage System (WISS) (Chou et al. 1985) offers basic DBMS functionality. At its top layer interface, WISS provides for (unstructured) records and scans of files of records, where scans can contain search predicates. All necessary additional functionality has to be implemented on top of this layer.

DASDBS (Darmstadt Database System) is another kernel system (Paul et al. 1987; Schek et al. 1990) that offers a general fixed interface. The reusable part of DASDBS is the complex record manager, which
handles record structures comparable to nested relations (Schek & Scholl 1986). DASDBS also supports multilevel transactions (Welkum 1991) and, therefore, provides support for implementing transaction management on upper levels. Object managers such as Bess (Biliris & Panagos 1995), EOS (Biliris 1992), the EXODUS storage manager (Carey et al. 1986), Kiosk (Nittel and Dittrich 1996), ObServer (Hornick & Zdonik 1987; Skarra et al. 1991), and Texas (Singhal et al. 1992) also fall into the class of kernel systems.

1.4.2 Pure Extensible Database Systems

Pure extensible database systems allow new parts such as abstract data types or index structures to be added to the system (note that the term “extensible database system” in the broader sense often refers to systems that support any kind of enhancing, extending, or customizing DBMS; Carey & Haas 1990).

Enhancing DBS with new Abstract Data Type (ADT) or index structures has been pioneered in the Ingres/Postgres systems (Stonebraker et al. 1983; Stonebraker 1986; Lynch & Stonebraker 1988). Ingres supports the definition of new ADTs, including operators. References to other tuples can be expressed through queries (i.e., the data type postquel), but otherwise ADTs and their associated relations still must be in first normal form. This restriction has been relaxed in systems that have a more powerful type system (e.g., an object-oriented data model) (Bancilhon et al. 1992; Dadam et al. 1986; Dittrich et al. 1986; Linnemann et al. 1988; Schek et al. 1990).

Another area in which extensions have been extensively considered are index structures. In Ingres/Postgres, existing indexes (such as B-trees) can be extended to also support new types (or support existing types in a better way). To extend an index mechanism, new implementations of type-specific operators of indexes have to be provided by the user. In this way, existing index structures are tailored to fit new purposes and thus have been called extended secondary indexes. Since most of the implementation of an index does not need to be changed, extensions are easier to perform than implementing a completely new index structure. The principle of extended secondary indexes has recently been applied in the DB2 UDB object-relational DBMS (Chen et al. 1999).

1.4.3 Customizable Systems

This kind of system is based on a complete DBMS that is modified or extended so that it satisfies new requirements. The basic DBMS is customized to a concrete, new DBMS. In principle, (internal) DBMS components are exchanged in order to achieve specific functionality in a
different way than in the original system. Therefore, a crucial issue is the underlying architecture and the proper definition of places where exchanges can be performed.

Starburst (Haas et al. 1990; Lindsay et al. 1987) is an example of a customizable DBMS. Its query language can be extended by new operators on relations (Haas et al. 1989). Various phases of query processing in Starburst are also customizable. Queries are internally represented as query graphs, and query rewrite transforms these graphs into equivalent, better ones. The rewrite part of Starburst’s query processor can be customized (Pirahesh et al. 1992) by adding new rewrite rules (where each rule is defined in the form of two C procedures). The Starburst query optimizer maps a query graph into an executable plan in a bottom-up manner. For each element of the query graph, it creates one or more alternative plans and selects the cheapest plan that meets the required properties. The mapping of query graph elements into plan operators is defined by Strategy Alternative Rules (STARS) (Lohman 1988). The optimizer can be customized by defining new STARS.

Storage methods can also be customized in Starburst in that (new) relation storage methods are plugged into the system. Relation storage methods implement the storage of relation instances and operations on them. They determine how a specific relation is represented physically (e.g., as a temporary relation or in the leaves of a B-tree). Furthermore, attachment types can be associated with relation instances. The meaning of attachments is that their operations are invoked as a side effect of relation-modification operations (operations that update a relation). Attachment types include access structures, integrity constraints, and triggers. Both relation storage methods and attachment types have to implement a collection of generic operations.

1.4.4 Toolkit Systems

Toolkit systems offer libraries of modules that implement alternative techniques for given tasks (e.g., physical access paths). The variable part of a DBMS is then built by selecting one technique from each library, and plugging together the chosen techniques.

The EXODUS (Carey et al. 1990, 1991) approach (see also Section 1.4.6) applies the idea of a toolkit for specific parts of the DBMS. A library is provided for access methods. While the library initially contains type-independent access methods such as B-trees, grid files, and linear hashing, it can also be extended by a DBI with new methods. Hereby, the DBI can use the DBMS implementation language E (Richardson & Carey 1986), a derivative of C++, for the implementation of extensions as well as for other parts of a DBMS. Furthermore, another library exists for operator methods, each of which implements an operation on a single type of storage object (e.g., selection). These operator
methods are used later to realize operators of a query language (see later sections).

The Open OODB (Open Object-Oriented Database) approach (Blakely 1994; Wells et al. 1992) supports the construction of object-oriented DBMSs. Open OODB distinguishes a meta-architecture and an extensible collection of modules implementing specific functionality (policies). The meta-architecture defines a set of kernel modules, mechanisms to define the system architecture (boundaries between components), and so forth. For some specific functional tasks, various policies can be applied (and can even be exchanged dynamically). Each domain for which multiple policies can be used is controlled by a policy manager, and all the policies of a specific domain are required to guarantee the same invariants (which ensures that they are interchangeable). In a nutshell the construction process with Open OODB consists of two major steps: defining the architecture with the means provided by the meta-architecture, and selecting policies for those required domains that allow multiple policies.

Further examples of toolkits are Trent (Unland & Schlageter 1992) for the construction of transaction managers (mainly, transaction structures and concurrency control) and A la carte (Drew et al. 1992) for the construction of heterogeneous DBMSs.

One problem in any toolkit approach is the consistency (or compatibility) of reused components. Lienert (1987) investigates conditions under which DBMSs can be configured. He first identifies the major tasks of a DBMS and generally distinguishes access (storing and retrieving data) from management (concurrency control, recovery, integrity enforcement, etc.). Furthermore, he elaborates the definition of standard techniques for these tasks and interrelationships between these techniques. Then, attributes representing certain properties of techniques are derived for the various tasks, rules are specified for deriving (deducing) some of them, and conditions are specified that have to hold for a configuration (e.g., a configuration is not correct if specific techniques have mutually contradictory values for some attributes).

1.4.5 Transformational Systems

In a transformational approach to DBMS construction, the DBI uses languages for the specification of the functions to be implemented. These functions are implemented using the interfaces of a lower layer, and DBI must also specify the mapping to that implementation base.

GENESIS (Batory et al. 1988a, 1988b) is a transformational approach that supports the implementation of data models as a sequence of layers. The interface of each layer defines its notions of files, records, and links between files. The implementation of a layer is described by the transformation of its concepts to those of the next-lower layer. Transformations themselves are collected in libraries, so that they can
be reused for future layer implementations. The basic (fixed) component is the JUPITER system for file management. The sequence of transformations maps the data-model concepts to the JUPITER interface.

The approach of GENESIS has been generalized to a construction method for hierarchical software systems (Batory & O’Malley 1992). The underlying assumption is that DBMSs can be constructed as layered systems. The central notion of this approach is the component, where each component is an element of a realm. All the elements of a realm have the same interface, but possibly different implementations. Then, a software system is described as a kind of algebraic expression. Component reuse refers to referencing the same component in multiple expressions.

Another transformational approach that uses specification constructs similar to those of Acta (Chrysanthis & Ramamritham 1994) has been described by Georgakopoulos et al. (1994). The transaction specification and management environment (TSME) consists of two building blocks: a transaction specification facility (TSF) and a programmable transaction management mechanism (TMM). TSF also uses the notion of dependency between transactions. Specific dependencies are implemented through Event-Condition-Action (ECA) rules in DOMS.

1.4.6 Generators

Generator systems support the specification of (parts of) a DBMS functionality and the generation of DBMS components based on those specifications. The DBI specifies a model (e.g., an optimizer, a data model, or a transaction model), which is input to a generator. The generator then automatically creates a software component that implements the specified model based on some implementation base (e.g., a storage manager or kernel in the case of data-model software generation). The knowledge for mapping the concepts specified in the model to the implementation base is in the generator.

An example of a generator system is the EXODUS query-optimizer generator (Graefe & DeWitt 1987). Input to the generator is a model description file, which contains a set of operators, a set of methods to be considered for constructing query execution plans, transformation rules, and implementation rules. Transformation rules specify legal transformations of query trees into new ones, and implementation rules define correspondences between operators and methods (i.e., which method can be used to implement an operator); for example the join operator could have as a corresponding method a hash join. In addition to the model description, a set of C procedures has to be supplied, which, for example, determine the cost functions of the various methods. Given this information, the generator can create a query optimizer for a DBMS under construction.
Volcano (Graefe & McKenna 1993), the successor of the EXODUS optimizer generator, also falls into the group of generator systems. Volcano has been used to build the optimizer for Open OODB (Blakeley et al. 1993).

1.4.7 Frameworks

Frameworks model families of software systems with a common prescribed architecture and behavior. They model the variable parts of the considered systems as abstract classes, and a concrete system can be built by deriving new subclasses from the abstract ones (called reuse by inheritance).

Opt++ (Kabra & DeWitt 1999) is a query-optimizer framework. Abstract classes model operator trees and access plans. Further classes model the transformations of operator trees into other trees or into access plans, or of access plans into other plans. Finally, search strategies are also represented as classes. The Opt++ framework defines a general architecture of a query optimizer in terms of instances of these classes. A concrete optimizer is built by deriving concrete classes from the abstract classes prescribed by the framework.

Other frameworks for building query optimizers are the ones described in Özsu et al. (1995), Cascades (Graefe 1995), and EROC (Extensible Reusable Optimization Components) (McKenna et al. 1996). Framboise (Fritschi et al. 1998) is a framework for layering active database functionality on top of passive DBMSs.

Generalized search trees (GiST) (Hellerstein et al. 1995) allow the incorporation of new index structures into a DBMS. GiST supports tree-based indexes; their general behavior (e.g., insertion and deletion) is predefined and thus does not need to be rewritten for new instances of GiST. New index structures can be built by implementing or overriding six methods specific for concrete search trees. The GiST approach has been extended to cover a broader spectrum of search trees and search techniques (e.g., nearest neighbor) (Aoki 1998). Kornacker et al. (1997) shows how concurrency control and recovery can be implemented for instances of GiST. GiST has recently been incorporated into Informix Dynamic Server with Universal Data Option (Kornacker 1999).

1.4.8 Discussion

Many of the techniques surveyed in this section have made their way into products (user-definable types, extensible index structures, etc.). While there are some success stories (i.e., techniques having found their way into products), there are also lessons to be learned and problems to be addressed.
An interesting observation is that recent efforts have assumed a
fixed data model, such as the object-relational or object-oriented
model. This is in contrast to older work, in which the implementation
of new data models was also investigated. The reasons for this trend
are, in our opinion, that there is no significant market for new special-
ized data models and that the invention of new data models requires
that most of the other DBMS parts (such as the query processor) as
well as tools (e.g., for database design) be adapted accordingly.

Most of the recent efforts also assumed a fixed transaction model
(e.g., ACID-transactions or closed nested transactions), while signifi-
cantly less work has been done on extensible transaction management.
Exceptions to this observation are concurrency control and recovery
for extensible index structures (e.g., Kornacker et al. 1997). The probable
reasons for this trend are that many of the proposed nonstandard
transaction models have never been implemented and questions
remain concerning their semantics and range of applicability.

Furthermore, it can be concluded from the work done in the past
and the functionality available today that some techniques are more
feasible than others or that they are well suited for specific cases. For
instance, techniques requiring a large implementation effort (e.g.,
implementing a DBMS on top of object managers) are only practical for
a vendor who wants to offer a suite of products and wants to share part
of the codebase among products. Customization techniques require a
sound understanding of the systems and of the effects that customiza-
tions have. They are therefore also only applicable for vendors, unless
their specification is possible in a very high-level way and implement-
ors are not required to know the internals of the core system.

A further noteworthy observation is that approaches in extensible
database systems seem to follow trends in software engineering, but
delayed a few years. For instance, while older approaches have used the
principles of generators, more recent efforts have devised framework-
based solutions.

Finally, another lesson is that feasible approaches to extensibility
should consider the entire DBMS. Approaches concentrating on a sin-
gle aspect (e.g., transaction management) tend to make assumptions
that restrict other parts of the DBMS. These restrictions then raise
questions of consistency and compatibility when other parts of the sys-
tem should be kept extensible or customizable as well. In consequence,
in order to be feasible, extension mechanisms in one area should not
preclude extensions in other parts of the system.

Finally, as more and more extensions become available, we expect
that problems known from software-reuse research and practice (Krue-
ger 1992) need to be solved in our context as well. For instance, when-
ever there is a significant number of components to choose from, users
need support for selection (i.e., finding the adequate reusable software
In cases in which extensions cannot be reused as is, users need help to adapt reused extensions.

We conclude from these lessons that feasible approaches to extensibility and customizability need to rely on the componentization of DBMSs. In this way, variable parts are clearly identified, as are the component plugs used for extension and composition. This in turn significantly reduces the amount of knowledge about internals required for extensions and customizations. The notion of component used to describe possible extensions is suitable for addressing reuse problems, due mainly to their higher level of abstraction, but also due to the requirement that connections (to other system parts) be well defined.

1.5 Component Database Models

In this section, we present the various types of CDBMSs. We consider two dimensions:

- Components: What kinds of components are considered? Which kind of database functionality or DBMS task can be represented as a component? How are components defined?
- Architecture: What is the generic DBMS architecture that allows plug-in components? What are the respective fixed and variable parts? How are components and connections described?

The classification given in this section does not imply that all the categories are completely disjoint. In fact, a concrete system can belong to multiple categories, for instance, if it allows the addition of different kinds of components. We return to this issue in Section 1.5.5.

1.5.1 Plug-in Components

The first category of CDBMSs comprises universal servers. The core system in this group is formed by a fully functional DBMS that implements all the standard functionality expected from a DBMS. Nonstandard features or functionality not yet supported can be plugged into this DBMS (see Figure 1.2). Thus, such a system is functionally complete and meets basic requirements, while extensions add further functionality for specialized needs.

The components in this kind of CDBMS are typically families of base and abstract data types or implementations of some DBMS function, such as new index structures. The DBMS architecture, among others, defines a number of plugs that components can use, for example, interfaces of functions that the DBMS will invoke and that the component thus must implement. In other words, the architecture formulates
expectations concerning interfaces that the component must meet in order to be integrated successfully.

To date, all systems in this category are based on the relational data model and existing relational DBMSs, and all of them offer some object-oriented extensions. We thus discuss this type of CDBMS in an object-relational (Stonebraker et al. 1999) context, although componentization is also possible for object-oriented database systems. Example systems include IBM’s DB2 UDB (IBM 1995), Informix Universal Server (Informix 1998), Oracle8 (Oracle 1999a), and Predator (Seshadri 1998). Descriptions of sample component developments can be found in (Blujuute et al. 1999; Deßloch & Mattos 1997).

The approaches we consider here aim at providing data management facilities for new nonstandard data types and for nonstandard or specialized database functionality within the DBMS. Instances of these new data types are thus stored in the database, and their manipulation and retrieval is implemented by the DBMS.

Assume an application needs support for data types not yet supported by the DBMS in use (e.g., spatial data or social security numbers). The ultimate task is to teach the DBMS how to store, manipulate, and retrieve instances of these data types.

In the first step, the designer has to model the structure of the desired new data types as well as their type-specific behavior. From a user’s point of view, new data types are either complex or atomic. Complex data types possess structure and their values can thus be represented as specialized records or tuples or collections using the data definition language. Atomic data types do not have an internal structure and consist of literal values (such as numbers or characters). For atomic types, the DBMS needs basic information, such as their length in bytes, in order to store their instances.

Thus, for spatial data, we might specify points as a complex data type modeling locations in three-dimensional space. 3DPoint could be specified as a tuple type with attributes x, y, and z, each of which is of type decimal. Another example would be Region, whose structure...
1.5 Component Database Models

could be defined as a pair of points \texttt{LowerLeft} and \texttt{UpperRight}. Social security numbers would be defined as an atomic type whose instances are 9 bytes long.

In addition to the structure of data types, the type-specific behavior of the new sorts of data needs to be specified. For each complex data type, its specific functions, operators, and predicates must be made known to the DBMS. In our example, a possible function would be the \texttt{move} function for points, and a typical predicate would be \texttt{overlaps}, which tests whether two regions intersect in some subregion.

Atomic types normally do not exhibit type-specific behavior. They, however, often require special treatment with respect to ordering and representation. Indeed, one reason to introduce a new type for non-standard atomic type is that existing DBMSs do not know how to sort them correctly. Thus, for new atomic types it is necessary to define operators such as \texttt{<}. Furthermore, the internal representation usually is not very telling for end users; thus, functions converting elements of atomic types from an internal (stored) representation to and from an external one are needed.

For each function, operator, and predicate, a signature (i.e., its name, arguments, and result) must be defined and an implementation must be provided. The implementation language in turn depends on the DBMS, possibilities ranging from DBMS-specific languages such as Oracle’s PL/SQL to general-purpose programming languages such as C or C++ or Java.

The collection of data types (their definition and implementation) forms a significant part of a component, which then needs to be plugged into the DBMS. To this end, DBMSs in this category offer a facility to register new components. Component registration introduces new definitions (for types, functions, etc.), and also informs the DBMS where (i.e., in which files) implementations can be found.

After a data type has been registered, applications can, in principle, start to use them (i.e., to create and retrieve instances of them). However, efficient retrieval and processing might require further enhancements to the DBMS, particularly to the access path manager and the query optimizer. Thus, we observe that extending a DBMS by plugging in new components often has a sort of domino effect because other parts must be adapted accordingly.

Current DBMSs typically contain B-tree access paths and possibly also hash-based indexes. B-trees can handle one-dimensional keys very well and rely on the orderability of keys. New types such as spatial data types can, however, be multidimensional; thus they would not be adequately served by B-trees, and, consequently, query processing might easily become inefficient. Therefore, in some situations it will be desirable to add new access methods to the DBMS, such as one that supports multidimensional indexing (Gaede & Günther 1998). In order to integrate well with other parts of the DBMS, such a new index has to
implement exactly those functions the DBMS calls for its indexes (i.e.,
insertion and deletion of entries, index scanning, etc.).

Furthermore, the addition of new data types also affects query pro-
cessing, in particular query optimization. Cost-based optimization tech-
niques, for instance, need information about the cost of each operator
(in terms of CPU consumption and I/O-operations) to find an optimal
plan. Thus, to ensure efficient query processing in this case, it is neces-
sary to provide the optimizer with the adequate information, such as
the knowledge of how it can estimate the cost of evaluating newly
defined predicates.

As a typical example of the aforementioned domino effect, consider
concurrency control for access paths. Many DBMSs use specialized
concurrency control protocols on indexes to prevent unnecessary lock-
ing conflicts (Bayer & Schkolnick 1997; Kornacker et al. 1997), which
otherwise would increase lock contention and decrease transaction
throughput. Therefore, whenever a new index is introduced, concur-
rency control (for this new index type) should also be adapted, which
is, however, not possible in current systems.

1.5.2 Database Middleware

The typical aim of systems falling into this category is to integrate exist-
ing data stores, that is, to leave data items under the control of their
original (external) management systems while integrating them into a
common DBMS-style framework. For instance, existing data stores
should be integrated into query processing or transaction management
of the entire system. External systems will in many cases exhibit differ-
ent capabilities, such as query languages with varying power or no
querying facilities at all. The different data stores might also have dif-
ferent data models (i.e., different data definition and structuring
means), or no explicit data model at all. Users and applications should,
nevertheless, be shielded from this kind of heterogeneity and should be
provided with a uniform and integrated view of the entire system. This
task is accomplished by the CDBMS acting as middleware (Orfali et al.
1996; Ferreira Rezende & Hergula 1998) between these data stores and
the applications of the integration. The overall aim of systems in this
group is similar to that of multidatabase systems (Sheth & Larson 1990;
Elmagarmid et al. 1999), although the latter typically consider only the
integration of database systems, instead of any kind of data store.

The goal of graceful integration is achieved through componentiza-
ton in the following way (Figure 1.3). The architecture introduces a
common (intermediate) format into which the local data formats can
be translated. Components are introduced that are able to perform this
kind of translation. Second, common interfaces and protocols define
how the database middleware system and the components interact
(e.g., in order to retrieve data from a data store). These components
(called wrappers) are also able to transform requests issued via these interfaces (e.g., queries) into requests understandable by the external system. In other words, these components implement the functionality needed to access from, within the database middleware system, data managed by the external data store.

The underlying problem in this respect is that the database middleware needs to understand the data formats and the functions of each data source. Two extreme alternatives exist to tackle this problem. In the first, information about the data sources’ interfaces are hard-wired into the (integrating) DBMS. The realm of integrable external data stores is thus restricted, and the DBMS needs to be extended for each specific type of data store. In the other alternative, a common data model, query language, or interface to external data stores is set as a prerequisite. For instance, we might require that all data stores understand SQL and be able to return the results of SQL queries in the form of relations. Each data store that does not implement SQL right away would thus have to be extended to do so. Moreover, all the data sources and the middleware would have to agree on one specific SQL dialect.

The solution that helps to overcome the intrinsic problems of both approaches lies in the introduction of additional components. In a nutshell, a component is pushed between the DBMS and each data source. These components serve to homogenize differences in formats and functionality from the DBMS’s point of view. From the data sources’ perspective, they level the different data source capabilities to a common basis. Thus, each component mediates between the data sources and the DBMS, or—from the DBMS’s point of view—wraps the data source.

The first prerequisite of this approach is a common data abstraction (e.g., objects). Second, the mediation components must offer a common interface. This interface is used by the DBMS to request data from...
the data sources. Each component should at least support the minimal interface, such as scanning a collection of data entities. Depending on the data source capabilities, its mediation component can, however, contribute more features or specialized functions, such as predicate evaluation. Whenever the DBMS executes a query and determines that it needs results from the data source, it sends a request to the corresponding component, which in turn translates the request into a form the data source can handle. Eventually, the component receives the results from the data source and converts them into the common format expected by the database middleware.

This approach requires an appropriately defined notion of components for wrapping the external data stores because it must match the requirements and characteristics of the DBMS while also using the capabilities of the data stores. Moreover, using the full potential of this approach means that a component is written once for each kind of data store (e.g., for a specific image management system) and used for all subsequent instances of the data store.

Ultimately, users should be allowed to introduce new components to integrate data stores not yet covered. To that end, the implementation of a component must be possible without the component implementor knowing the internal structure and operations of the database middleware; the requirements to be met by a component must be entirely expressed in its interface.

Examples of this approach include Disco (Tomasic et al. 1998), Garlic (Tork Roth & Schwartz 1997), OLE DB (Blakeley 1996a, 1996b; Microsoft 1996), Tsimmis (Papakonstantinou et al. 1995a), Harmony (Röhm & Böhm 1999) (which implements the CORBA query service), and Sybase Adaptive Server Enterprise (Olson et al. 1998) (which allows access to external data stores, in Sybase called specialty data stores, and other types of database systems).

1.5.3 **DBMS Services**

The third type of componentized DBMS is characterized by a service-oriented view of database functionality. All DBMS and related tasks are unbundled (Geppert & Dittrich 1998) into services. As a result, a monolithic DBMS is transformed into a set of stand-alone services. For instance, the unbundling process can result in persistence, query, and transaction services. Applications then no longer operate on full-fledged DBMSs, but instead use those services as needed (Figure 1.4).

Each service is defined by one or more interfaces and implemented by some software systems. Services (i.e., their interfaces) are defined in a common model or language. Services are implemented using a common platform in order to render the service implementations exchangeable and freely combinable. Services should be as indepen-
dent as possible from one another; that is, they should not rely on the availability of a specific other service in the environment and they should not assume that other services are implemented in a particular way.

In this scenario, (database) services and their implementations are viewed as components. Given that both the platform and the service interfaces are standardized, exchangeability and compatibility are achieved. Different implementations of a service can be exchanged, and implementations of different services—possibly from different vendors—can be plugged together.

An example of this approach are CORBA services (OMG 1995a), which leverage several DBMS tasks to general object systems. These services are standardized by the Object Management Group (OMG). Service interfaces are defined using the Interface Definition Language. Services related to database functionality include persistency, concurrency, and queries. Such services are implemented on top of Object Request Brokers (ORB) (OMG 1997a).

In contrast to the other approaches discussed here, the components (i.e., services) are not meant to extend or customize a DBMS. Rather, the systems constructible by using services are distributed applications located above the DBMS level. In fact, services such as persistence could be implemented by a DBMS, and the transaction service might be implemented by transaction processing monitors (Bernstein & Newcomer 1996).

The underlying semantics and models of services (such as the transaction model or query language) are fixed. Thus, for a transaction service, there will be distinct implementations all implementing transactions such as flat ACID transactions, but transactions such as cooperative or other forms of nonstandard transactions (Elmagarmid 1992) will not be supported.

A second example of this approach is the strawman architecture developed at Computer Corporation of America (CCA 1982). The aim of this study was to propose standard interfaces between users or applications and DBMSs as well as standards for internal interfaces (such that different DBMS subsystems can be combined more easily). This
A study identified 79 subcomponents, which are grouped together into 38 components, some of which denote internal functions, while others refer to external (i.e., visible at the DBMS interface) ones. The subcomponents are partitioned into six groups of related tasks. For each subcomponent, procedures and interfaces are proposed. Therefore, the view of DBMS architecture is more service-oriented, and concrete components are proposed to implement such services (to the best of our knowledge, however, a DBMS implementing this architecture has never been built).

1.5.4 Configurable DBMS

In the previous form of CDBMSs, the set of services have been standardized and fixed. One step further are configurable DBMSs that allow new DBMS parts to be developed and integrated into a DBMS (Figure 1.5). Thus, configurable DBMSs are similar to DBMS services in that they also rely on unbundled DBMS tasks that can be mixed and matched to obtain database support. The difference lies in the possibility of adapting service implementations to new requirements or of defining new services whenever needed. The components are DBMS subsystems, which are defined in an underlying architecture model. In this approach, the architecture model is also used to model the DBMS architecture, which is no longer fixed.

Configurable DBMSs also consider services as unbundled representations of DBMS tasks. However, the models underlying the various services and defining the semantics of the corresponding DBMS parts can now in addition be customized. As a consequence, components for the same DBMS task can vary not only in their implementations for the same standardized interface, but also in their interfaces for the same task. DBMS implementors select (or construct new) components implementing the desired functionality and obtain a DBMS by assembling the selected components (Figure 1.5). The DBMS is thus the result of a configuration process.

An example of a configurable DBMS is the KIDS project (Geppert & Dittrich 1995; Geppert et al. 1997), which aims at constructing a DBMS by developing subsystems that implement various aspects of a DBMS (such as transaction management or constraint enforcement) and by then configuring these subsystems together into a coherent and complete DBMS.

The underlying architecture model provides for constructs that are adequate for defining the architecture of DBMSs. The tasks and functionality of a DBMS and its components are modeled by means of services. Services are provided by reactive components called brokers (i.e., brokers are responsible for services). In the case of service requests, the responsible brokers react by providing the service. The functionality of
each subsystem under construction is represented as a set of services, and one or more brokers are designated as components implementing the subsystem.

The construction process defines how to proceed in order to obtain a DBMS with the desired functionality. This process consists of several phases including requirements analysis, design, implementation, and integration of multiple DBMS subsystems. Some phases of the process are common to all constructible DBMSs and are independent of subsystems (e.g., requirements analysis and architecture design). For each type of subsystem, a dedicated construction subprocess is defined and integrated into the enclosing DBMS-construction process. For each subsystem, a dedicated specification language is used to define its functionality (such as Acta in the case of transaction models, Chrysanthis & Ramamritham 1994; or Second-order Signature (SOS) in the case of data models, Güting 1993). These specifications serve as the input to subsystem-specific implementation phases, which in turn use techniques such as the generation of subsystems or the configuration of subsystems out of reusable, already existing components (Geppert & Dittrich 1995).

1.5.5 Discussion

We now summarize and discuss the elaboration of CDBMS models. Table 1.1 summarizes the characteristics of the four categories. These categories are not necessarily disjoint. For instance, it is conceivable that both plug-in components for nonstandard data and wrappers for accessing external data stores can be added to a single system. Such a system would therefore belong to the first two categories (e.g., as outlined in Stonebraker et al. 1998). Likewise, OLE DB could also be classified as a
configurable DBMS, since it in principle allows the exchange and addition of internal components, for example, to add specialized query processors. In the remainder of this book, representatives of these groups are discussed in more detail.

### 1.6 Summary and Conclusion

Today’s users of database technology require extensibility in all conceivable forms. In order to maintain the software quality and robustness that current (monolithic) DBMS engines exhibit, yet also meet the extensibility requirements, database technology needs to adopt the principles of component technology.

This chapter has also classified approaches to componentizing DBMS and database functionality in general. The following chapters describe prominent representatives of these classes in more detail.