M. Tamer Özsu
Patrick Valduriez

Principles of Distributed Database Systems

Third Edition

Springer

Principles of Distributed Database Systems

M. Tamer Özsu • Patrick Valduriez

Principles of Distributed Database Systems

Third Edition

Springer

M. Tamer Özsu
David R. Cheriton School of
Computer Science
University of Waterloo
Waterloo Ontario
Canada N2L 3G1
Tamer.Ozsu@uwaterloo.ca

Patrick Valduriez
INRIA
LIRMM
161 rue Ada
34392 Montpellier Cedex
France
Patrick.Valduriez@inria.fr

This book was previously published by: Pearson Education, Inc.

ISBN 978-1-4419-8833-1 e-ISBN 978-1-4419-8834-8
DOI 10.1007/978-1-4419-8834-8
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2011922491

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

To my family
and my parents
M.T.Ö.

To Esther, my daughters Anna, Juliette and Sarah, and my parents
P.V.

Preface

It has been almost twenty years since the first edition of this book appeared, and ten years since we released the second edition. As one can imagine, in a fast changing area such as this, there have been significant changes in the intervening period. Distributed data management went from a potentially significant technology to one that is common place. The advent of the Internet and the World Wide Web have certainly changed the way we typically look at distribution. The emergence in recent years of different forms of distributed computing, exemplified by data streams and cloud computing, has regenerated interest in distributed data management. Thus, it was time for a major revision of the material.

We started to work on this edition five years ago, and it has taken quite a while to complete the work. The end result, however, is a book that has been heavily revised – while we maintained and updated the core chapters, we have also added new ones. The major changes are the following:

1. Database integration and querying is now treated in much more detail, reflecting the attention these topics have received in the community in the past decade. Chapter 4 focuses on the integration process, while Chapter 9 discusses querying over multidatabase systems.
2. The previous editions had only brief discussion of data replication protocols. This topic is now covered in a separate chapter (Chapter 13) where we provide an in-depth discussion of the protocols and how they can be integrated with transaction management.
3. Peer-to-peer data management is discussed in depth in Chapter 16. These systems have become an important and interesting architectural alternative to classical distributed database systems. Although the early distributed database systems architectures followed the peer-to-peer paradigm, the modern incarnation of these systems have fundamentally different characteristics, so they deserve in-depth discussion in a chapter of their own.
4. Web data management is discussed in Chapter 17. This is a difficult topic to cover since there is no unifying framework. We discuss various aspects

of the topic ranging from web models to search engines to distributed XML processing.

5. Earlier editions contained a chapter where we discussed "recent issues" at the time. In this edition, we again have a similar chapter (Chapter 18) where we cover stream data management and cloud computing. These topics are still in a flux and are subjects of considerable ongoing research. We highlight the issues and the potential research directions.

The resulting manuscript strikes a balance between our two objectives, namely to address new and emerging issues, and maintain the main characteristics of the book in addressing the principles of distributed data management.

The organization of the book can be divided into two major parts. The first part covers the fundamental principles of distributed data management and consist of Chapters 1 to 14. Chapter 2 in this part covers the background and can be skipped if the students already have sufficient knowledge of the relational database concepts and the computer network technology. The only part of this chapter that is essential is Example 2.3, which introduces the running example that we use throughout much of the book. The second part covers more advanced topics and includes Chapters 15 – 18. What one covers in a course depends very much on the duration and the course objectives. If the course aims to discuss the fundamental techniques, then it might cover Chapters 1, 3, 5, 6–8, 10–12. An extended coverage would include, in addition to the above, Chapters 4, 9, and 13. Courses that have time to cover more material can selectively pick one or more of Chapters 15 – 18 from the second part.

Many colleagues have assisted with this edition of the book. S. Keshav (University of Waterloo) has read and provided many suggestions to update the sections on computer networks. Renée Miller (University of Toronto) and Erhard Rahm (University of Leipzig) read an early draft of Chapter 4 and provided many comments, Alon Halevy (Google) answered a number of questions about this chapter and provided a draft copy of his upcoming book on this topic as well as reading and providing feedback on Chapter 9, Avigdor Gal (Technion) also reviewed and critiqued this chapter very thoroughly. Matthias Jarke and Xiang Li (University of Aachen), Gottfried Vossen (University of Muenster), Erhard Rahm and Andreas Thor (University of Leipzig) contributed exercises to this chapter. Hubert Naacke (University of Paris 6) contributed to the section on heterogeneous cost modeling and Fabio Porto (LNCC, Petropolis) to the section on adaptive query processing of Chapter 9. Data replication (Chapter 13) could not have been written without the assistance of Gustavo Alonso (ETH Zürich) and Bettina Kemme (McGill University). Tamer spent four months in Spring 2006 visiting Gustavo where work on this chapter began and involved many long discussions. Bettina read multiple iterations of this chapter over the next one year criticizing everything and pointing out better ways of explaining the material. Esther Pacitti (University of Montpellier) also contributed to this chapter, both by reviewing it and by providing background material; she also contributed to the section on replication in database clusters in Chapter 14. Ricardo Jimenez-Peris also contributed to that chapter in the section on fault-tolerance in database clusters. Khuzaima Daudjee (University of Waterloo) read and provided

comments on this chapter as well. Chapter 15 on Distributed Object Database Management was reviewed by Serge Abiteboul (INRIA), who provided important critique of the material and suggestions for its improvement. Peer-to-peer data management (Chapter 16) owes a lot to discussions with Beng Chin Ooi (National University of Singapore) during the four months Tamer was visiting NUS in the fall of 2006. The section of Chapter 16 on query processing in P2P systems uses material from the PhD work of Reza Akbarinia (INRIA) and Wenceslao Palma (PUC-Valparaiso, Chile) while the section on replication uses material from the PhD work of Vidal Martins (PUCPR, Curitiba). The distributed XML processing section of Chapter 17 uses material from the PhD work of Ning Zhang (Facebook) and Patrick Kling at the University of Waterloo, and Ying Zhang at CWI. All three of them also read the material and provided significant feedback. Victor Muntés i Mulero (Universitat Politècnica de Catalunya) contributed to the exercises in that chapter. Özgür Ulusoy (Bilkent University) provided comments and corrections on Chapters 16 and 17. Data stream management section of Chapter 18 draws from the PhD work of Lukasz Golab (AT&T Labs-Research), and Yingying Tao at the University of Waterloo. Walid Aref (Purdue University) and Avigdor Gal (Technion) used the draft of the book in their courses, which was very helpful in debugging certain parts. We thank them, as well as many colleagues who had helped out with the first two editions, for all their assistance. We have not always followed their advice, and, needless to say, the resulting problems and errors are ours. Students in two courses at the University of Waterloo (Web Data Management in Winter 2005, and Internet-Scale Data Distribution in Fall 2005) wrote surveys as part of their coursework that were very helpful in structuring some chapters. Tamer taught courses at ETH Zürich (PDDBS – Parallel and Distributed Databases in Spring 2006) and at NUS (CS5225 – Parallel and Distributed Database Systems in Fall 2010) using parts of this edition. We thank students in all these courses for their contributions and their patience as they had to deal with chapters that were works-in-progress – the material got cleaned considerably as a result of these teaching experiences.

You will note that the publisher of the third edition of the book is different than the first two editions. Pearson, our previous publisher, decided not to be involved with the third edition. Springer subsequently showed considerable interest in the book. We would like to thank Susan Lagerstrom-Fife and Jennifer Evans of Springer for their lightning-fast decision to publish the book, and Jennifer Mauer for a ton of hand-holding during the conversion process. We would also like to thank Tracy Dunkelberger of Pearson who shepherded the reversal of the copyright to us without delay.

As in earlier editions, we will have presentation slides that can be used to teach from the book as well as solutions to most of the exercises. These will be available from Springer to instructors who adopt the book and there will be a link to them from the book's site at springer.com.

Finally, we would be very interested to hear your comments and suggestions regarding the material. We welcome any feedback, but we would particularly like to receive feedback on the following aspects:

1. any errors that may have remained despite our best efforts (although we hope there are not many);
2. any topics that should no longer be included and any topics that should be added or expanded; and
3. any exercises that you may have designed that you would like to be included in the book.

M. Tamer Özsu (Tamer.Ozsu@uwaterloo.ca)
Patrick Valduriez (Patrick.Valduriez@inria.fr)
November 2010

Contents

Chapter 1
Introduction

Distributed database system (DDBS) technology is the union of what appear to be two diametrically opposed approaches to data processing: *database system* and *computer network* technologies. Database systems have taken us from a paradigm of data processing in which each application defined and maintained its own data (Figure 1.1) to one in which the data are defined and administered centrally (Figure 1.2). This new orientation results in *data independence*, whereby the application programs are immune to changes in the logical or physical organization of the data, and vice versa.

One of the major motivations behind the use of database systems is the desire to integrate the operational data of an enterprise and to provide centralized, thus controlled access to that data. The technology of computer networks, on the other hand, promotes a mode of work that goes against all centralization efforts. At first glance it might be difficult to understand how these two contrasting approaches can possibly be synthesized to produce a technology that is more powerful and more promising than either one alone. The key to this understanding is the realization

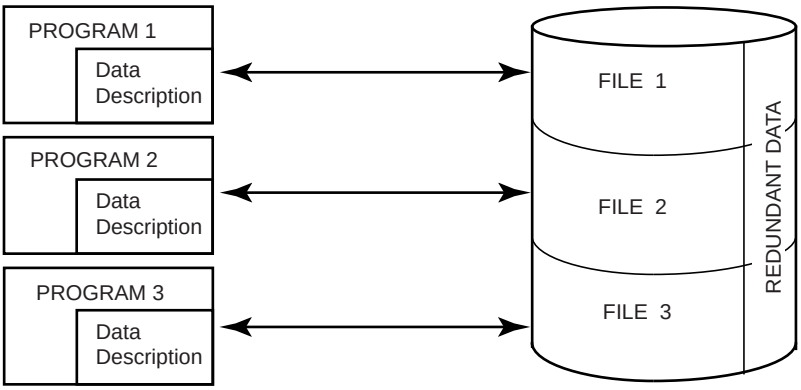

Fig. 1.1 Traditional File Processing

M.T. Özsu and P. Valduriez, *Principles of Distributed Database Systems: Third Edition*,
DOI 10.1007/978-1-4419-8834-8_1, © Springer Science+Business Media, LLC 2011

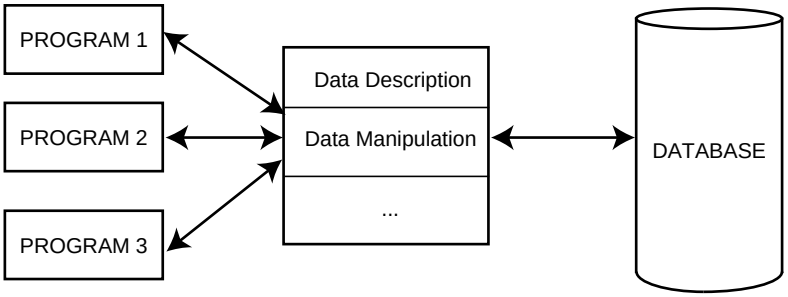

Fig. 1.2 Database Processing

that the most important objective of the database technology is *integration*, not *centralization*. It is important to realize that either one of these terms does not necessarily imply the other. It is possible to achieve integration without centralization, and that is exactly what the distributed database technology attempts to achieve.

In this chapter we define the fundamental concepts and set the framework for discussing distributed databases. We start by examining distributed systems in general in order to clarify the role of database technology within distributed data processing, and then move on to topics that are more directly related to DDBS.

1.1 Distributed Data Processing

The term *distributed processing* (or *distributed computing*) is hard to define precisely. Obviously, some degree of distributed processing goes on in any computer system, even on single-processor computers where the central processing unit (CPU) and input/output (I/O) functions are separated and overlapped. This separation and overlap can be considered as one form of distributed processing. The widespread emergence of parallel computers has further complicated the picture, since the distinction between distributed computing systems and some forms of parallel computers is rather vague.

In this book we define distributed processing in such a way that it leads to a definition of a distributed database system. The working definition we use for a *distributed computing system* states that it is a number of autonomous processing elements (not necessarily homogeneous) that are interconnected by a computer network and that cooperate in performing their assigned tasks. The "processing element" referred to in this definition is a computing device that can execute a program on its own. This definition is similar to those given in distributed systems textbooks (e.g., [Tanenbaum and van Steen, 2002] and [Colouris et al., 2001]).

A fundamental question that needs to be asked is: What is being distributed? One of the things that might be distributed is the *processing logic*. In fact, the definition of a distributed computing system given above implicitly assumes that the

processing logic or processing elements are distributed. Another possible distribution is according to *function*. Various functions of a computer system could be delegated to various pieces of hardware or software. A third possible mode of distribution is according to *data*. Data used by a number of applications may be distributed to a number of processing sites. Finally, *control* can be distributed. The control of the execution of various tasks might be distributed instead of being performed by one computer system. From the viewpoint of distributed database systems, these modes of distribution are all necessary and important. In the following sections we talk about these in more detail.

Another reasonable question to ask at this point is: Why do we distribute at all? The classical answers to this question indicate that distributed processing better corresponds to the organizational structure of today's widely distributed enterprises, and that such a system is more reliable and more responsive. More importantly, many of the current applications of computer technology are inherently distributed. Web-based applications, electronic commerce business over the Internet, multimedia applications such as news-on-demand or medical imaging, manufacturing control systems are all examples of such applications.

From a more global perspective, however, it can be stated that the fundamental reason behind distributed processing is to be better able to cope with the large-scale data management problems that we face today, by using a variation of the well-known divide-and-conquer rule. If the necessary software support for distributed processing can be developed, it might be possible to solve these complicated problems simply by dividing them into smaller pieces and assigning them to different software groups, which work on different computers and produce a system that runs on multiple processing elements but can work efficiently toward the execution of a common task.

Distributed database systems should also be viewed within this framework and treated as tools that could make distributed processing easier and more efficient. It is reasonable to draw an analogy between what distributed databases might offer to the data processing world and what the database technology has already provided. There is no doubt that the development of general-purpose, adaptable, efficient distributed database systems has aided greatly in the task of developing distributed software.

1.2 What is a Distributed Database System?

We define a *distributed database* as *a collection of multiple, logically interrelated databases distributed over a computer network*. A *distributed database management system* (distributed DBMS) is then defined as *the software system that permits the management of the distributed database and makes the distribution transparent to the users*. Sometimes "distributed database system" (DDBS) is used to refer jointly to the distributed database and the distributed DBMS. The two important terms in these definitions are "*logically interrelated*" and "*distributed over a computer network*." They help eliminate certain cases that have sometimes been accepted to represent a DDBS.

A DDBS is not a "collection of files" that can be individually stored at each node of a computer network. To form a DDBS, files should not only be logically related, but there should be structured among the files, and access should be via a common interface. We should note that there has been much recent activity in providing DBMS functionality over semi-structured data that are stored in files on the Internet (such as Web pages). In light of this activity, the above requirement may seem unnecessarily strict. Nevertheless, it is important to make a distinction between a DDBS where this requirement is met, and more general distributed data management systems that provide a "DBMS-like" access to data. In various chapters of this book, we will expand our discussion to cover these more general systems.

It has sometimes been assumed that the physical distribution of data is not the most significant issue. The proponents of this view would therefore feel comfortable in labeling as a distributed database a number of (related) databases that reside in the same computer system. However, the physical distribution of data is important. It creates problems that are not encountered when the databases reside in the same computer. These difficulties are discussed in Section 1.5. Note that physical distribution does not necessarily imply that the computer systems be geographically far apart; they could actually be in the same room. It simply implies that the communication between them is done over a network instead of through shared memory or shared disk (as would be the case with *multiprocessor systems*), with the network as the only shared resource.

This suggests that multiprocessor systems should not be considered as DDBSs. Although shared-nothing multiprocessors, where each processor node has its own primary and secondary memory, and may also have its own peripherals, are quite similar to the distributed environment that we focus on, there are differences. The fundamental difference is the mode of operation. A multiprocessor system design is rather symmetrical, consisting of a number of identical processor and memory components, and controlled by one or more copies of the same operating system that is responsible for a strict control of the task assignment to each processor. This is not true in distributed computing systems, where heterogeneity of the operating system as well as the hardware is quite common. Database systems that run over multiprocessor systems are called *parallel database systems* and are discussed in Chapter 14.

A DDBS is also not a system where, despite the existence of a network, the database resides at only one node of the network (Figure 1.3). In this case, the problems of database management are no different than the problems encountered in a centralized database environment (shortly, we will discuss client/server DBMSs which relax this requirement to a certain extent). The database is centrally managed by one computer system (site 2 in Figure 1.3) and all the requests are routed to that site. The only additional consideration has to do with transmission delays. It is obvious that the existence of a computer network or a collection of "files" is not sufficient to form a distributed database system. What we are interested in is an environment where data are distributed among a number of sites (Figure 1.4).

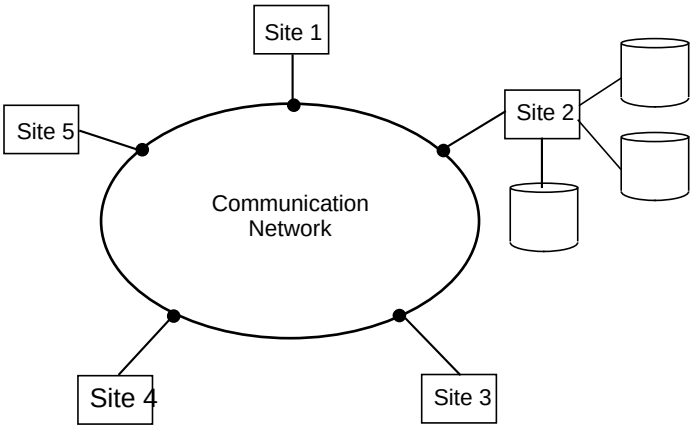

Fig. 1.3 Central Database on a Network

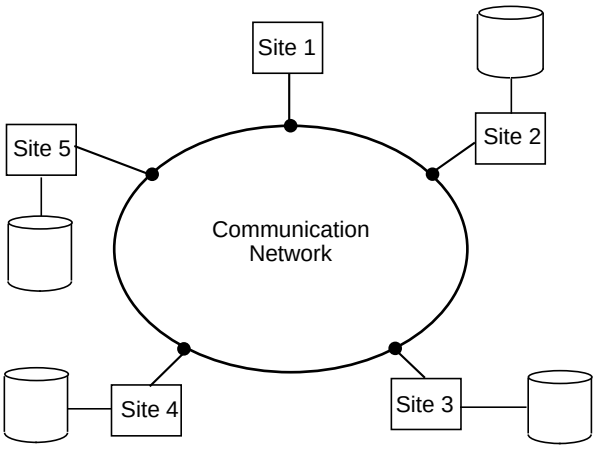

Fig. 1.4 DDBS Environment

1.3 Data Delivery Alternatives

In distributed databases, data are "delivered" from the sites where they are stored to where the query is posed. We characterize the data delivery alternatives along three orthogonal dimensions: *delivery modes*, *frequency* and *communication methods*. The combinations of alternatives along each of these dimensions (that we discuss next) provide a rich design space.

The alternative delivery modes are pull-only, push-only and hybrid. In the *pull-only* mode of data delivery, the transfer of data from servers to clients is initiated by a client pull. When a client request is received at a server, the server responds by locating the requested information. The main characteristic of pull-based delivery is that the arrival of new data items or updates to existing data items are carried out at a

server without notification to clients unless clients explicitly poll the server. Also, in pull-based mode, servers must be interrupted continuously to deal with requests from clients. Furthermore, the information that clients can obtain from a server is limited to when and what clients know to ask for. Conventional DBMSs offer primarily pull-based data delivery.

In the *push-only* mode of data delivery, the transfer of data from servers to clients is initiated by a server push in the absence of any specific request from clients. The main difficulty of the push-based approach is in deciding which data would be of common interest, and when to send them to clients – alternatives are periodic, irregular, or conditional. Thus, the usefulness of server push depends heavily upon the accuracy of a server to predict the needs of clients. In push-based mode, servers disseminate information to either an unbounded set of clients (random broadcast) who can listen to a medium or selective set of clients (multicast), who belong to some categories of recipients that may receive the data.

The hybrid mode of data delivery combines the client-pull and server-push mechanisms. The continuous (or continual) query approach (e.g., [Liu et al., 1996],[Terry et al., 1992],[Chen et al., 2000],[Pandey et al., 2003]) presents one possible way of combining the pull and push modes: namely, the transfer of information from servers to clients is first initiated by a client pull (by posing the query), and the subsequent transfer of updated information to clients is initiated by a server push.

There are three typical frequency measurements that can be used to classify the regularity of data delivery. They are *periodic*, *conditional*, and *ad-hoc* or *irregular*.

In periodic delivery, data are sent from the server to clients at regular intervals. The intervals can be defined by system default or by clients using their profiles. Both pull and push can be performed in periodic fashion. Periodic delivery is carried out on a regular and pre-specified repeating schedule. A client request for IBM's stock price every week is an example of a periodic pull. An example of periodic push is when an application can send out stock price listing on a regular basis, say every morning. Periodic push is particularly useful for situations in which clients might not be available at all times, or might be unable to react to what has been sent, such as in the mobile setting where clients can become disconnected.

In conditional delivery, data are sent from servers whenever certain conditions installed by clients in their profiles are satisfied. Such conditions can be as simple as a given time span or as complicated as event-condition-action rules. Conditional delivery is mostly used in the hybrid or push-only delivery systems. Using conditional push, data are sent out according to a pre-specified condition, rather than any particular repeating schedule. An application that sends out stock prices only when they change is an example of conditional push. An application that sends out a balance statement only when the total balance is 5% below the pre-defined balance threshold is an example of hybrid conditional push. Conditional push assumes that changes are critical to the clients, and that clients are always listening and need to respond to what is being sent. Hybrid conditional push further assumes that missing some update information is not crucial to the clients.

Ad-hoc delivery is irregular and is performed mostly in a pure pull-based system. Data are pulled from servers to clients in an ad-hoc fashion whenever clients request

it. In contrast, periodic pull arises when a client uses polling to obtain data from servers based on a regular period (schedule).

The third component of the design space of information delivery alternatives is the communication method. These methods determine the various ways in which servers and clients communicate for delivering information to clients. The alternatives are *unicast* and *one-to-many*. In unicast, the communication from a server to a client is one-to-one: the server sends data to one client using a particular delivery mode with some frequency. In one-to-many, as the name implies, the server sends data to a number of clients. Note that we are not referring here to a specific protocol; one-to-many communication may use a multicast or broadcast protocol.

We should note that this characterization is subject to considerable debate. It is not clear that every point in the design space is meaningful. Furthermore, specification of alternatives such as conditional **and** periodic (which may make sense) is difficult. However, it serves as a first-order characterization of the complexity of emerging distributed data management systems. For the most part, in this book, we are concerned with pull-only, ad hoc data delivery systems, although examples of other approaches are discussed in some chapters.

1.4 Promises of DDBSs

Many advantages of DDBSs have been cited in literature, ranging from sociological reasons for decentralization [D'Oliviera, 1977] to better economics. All of these can be distilled to four fundamentals which may also be viewed as promises of DDBS technology: transparent management of distributed and replicated data, reliable access to data through distributed transactions, improved performance, and easier system expansion. In this section we discuss these promises and, in the process, introduce many of the concepts that we will study in subsequent chapters.

1.4.1 Transparent Management of Distributed and Replicated Data

Transparency refers to separation of the higher-level semantics of a system from lower-level implementation issues. In other words, a transparent system "hides" the implementation details from users. The advantage of a fully transparent DBMS is the high level of support that it provides for the development of complex applications. It is obvious that we would like to make all DBMSs (centralized or distributed) fully transparent.

Let us start our discussion with an example. Consider an engineering firm that has offices in Boston, Waterloo, Paris and San Francisco. They run projects at each of these sites and would like to maintain a database of their employees, the projects and other related data. Assuming that the database is relational, we can store

this information in two relations: EMP(ENO, ENAME, TITLE)[1] and PROJ(PNO, PNAME, BUDGET). We also introduce a third relation to store salary information: SAL(TITLE, AMT) and a fourth relation ASG which indicates which employees have been assigned to which projects for what duration with what responsibility: ASG(ENO, PNO, RESP, DUR). If all of this data were stored in a centralized DBMS, and we wanted to find out the names and employees who worked on a project for more than 12 months, we would specify this using the following SQL query:

```
SELECT  ENAME, AMT
FROM    EMP, ASG, SAL
WHERE   ASG.DUR > 12
AND     EMP.ENO = ASG.ENO
AND     SAL.TITLE = EMP.TITLE
```

However, given the distributed nature of this firm's business, it is preferable, under these circumstances, to localize data such that data about the employees in Waterloo office are stored in Waterloo, those in the Boston office are stored in Boston, and so forth. The same applies to the project and salary information. Thus, what we are engaged in is a process where we partition each of the relations and store each partition at a different site. This is known as *fragmentation* and we discuss it further below and in detail in Chapter 3.

Furthermore, it may be preferable to duplicate some of this data at other sites for performance and reliability reasons. The result is a distributed database which is fragmented and replicated (Figure 1.5). Fully transparent access means that the users can still pose the query as specified above, without paying any attention to the fragmentation, location, or replication of data, and let the system worry about resolving these issues.

For a system to adequately deal with this type of query over a distributed, fragmented and replicated database, it needs to be able to deal with a number of different types of transparencies. We discuss these in this section.

1.4.1.1 Data Independence

Data independence is a fundamental form of transparency that we look for within a DBMS. It is also the only type that is important within the context of a centralized DBMS. It refers to the immunity of user applications to changes in the definition and organization of data, and vice versa.

As is well-known, data definition occurs at two levels. At one level the logical structure of the data are specified, and at the other level its physical structure. The former is commonly known as the *schema definition*, whereas the latter is referred to as the *physical data description*. We can therefore talk about two types of data

[1] We discuss relational systems in Chapter 2 (Section 2.1) where we develop this example further. For the time being, it is sufficient to note that this nomenclature indicates that we have just defined a relation with three attributes: ENO (which is the key, identified by underlining), ENAME and TITLE.

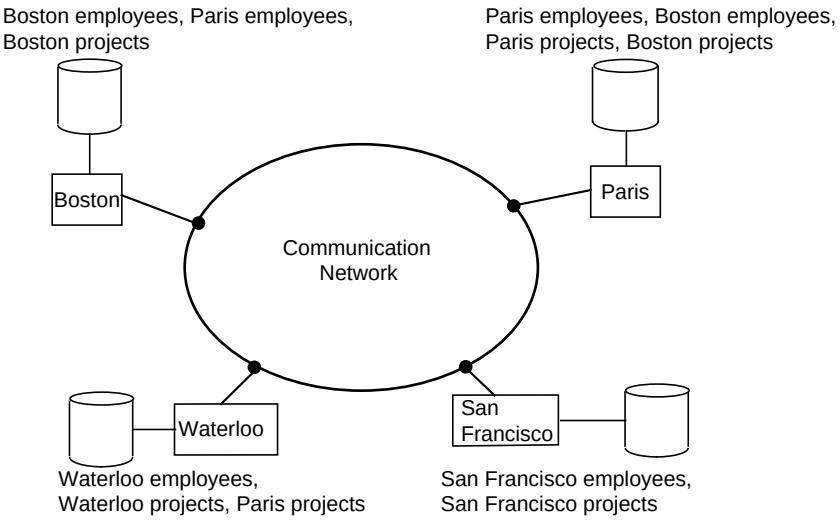

Fig. 1.5 A Distributed Application

independence: logical data independence and physical data independence. *Logical data independence* refers to the immunity of user applications to changes in the logical structure (i.e., schema) of the database. *Physical data independence*, on the other hand, deals with hiding the details of the storage structure from user applications. When a user application is written, it should not be concerned with the details of physical data organization. Therefore, the user application should not need to be modified when data organization changes occur due to performance considerations.

1.4.1.2 Network Transparency

In centralized database systems, the only available resource that needs to be shielded from the user is the data (i.e., the storage system). In a distributed database environment, however, there is a second resource that needs to be managed in much the same manner: the network. Preferably, the user should be protected from the operational details of the network; possibly even hiding the existence of the network. Then there would be no difference between database applications that would run on a centralized database and those that would run on a distributed database. This type of transparency is referred to as *network transparency* or *distribution transparency*.

One can consider network transparency from the viewpoint of either the services provided or the data. From the former perspective, it is desirable to have a uniform means by which services are accessed. From a DBMS perspective, distribution transparency requires that users do not have to specify where data are located.

Sometimes two types of distribution transparency are identified: location transparency and naming transparency. *Location transparency* refers to the fact that the

command used to perform a task is independent of both the location of the data and the system on which an operation is carried out. *Naming transparency* means that a unique name is provided for each object in the database. In the absence of naming transparency, users are required to embed the location name (or an identifier) as part of the object name.

1.4.1.3 Replication Transparency

The issue of replicating data within a distributed database is introduced in Chapter 3 and discussed in detail in Chapter 13. At this point, let us just mention that for performance, reliability, and availability reasons, it is usually desirable to be able to distribute data in a replicated fashion across the machines on a network. Such replication helps performance since diverse and conflicting user requirements can be more easily accommodated. For example, data that are commonly accessed by one user can be placed on that user's local machine as well as on the machine of another user with the same access requirements. This increases the locality of reference. Furthermore, if one of the machines fails, a copy of the data are still available on another machine on the network. Of course, this is a very simple-minded description of the situation. In fact, the decision as to whether to replicate or not, and how many copies of any database object to have, depends to a considerable degree on user applications. We will discuss these in later chapters.

Assuming that data are replicated, the transparency issue is whether the users should be aware of the existence of copies or whether the system should handle the management of copies and the user should act as if there is a single copy of the data (note that we are not referring to the placement of copies, only their existence). From a user's perspective the answer is obvious. It is preferable not to be involved with handling copies and having to specify the fact that a certain action can and/or should be taken on multiple copies. From a systems point of view, however, the answer is not that simple. As we will see in Chapter 11, when the responsibility of specifying that an action needs to be executed on multiple copies is delegated to the user, it makes transaction management simpler for distributed DBMSs. On the other hand, doing so inevitably results in the loss of some flexibility. It is not the system that decides whether or not to have copies and how many copies to have, but the user application. Any change in these decisions because of various considerations definitely affects the user application and, therefore, reduces data independence considerably. Given these considerations, it is desirable that replication transparency be provided as a standard feature of DBMSs. Remember that replication transparency refers only to the existence of replicas, not to their actual location. Note also that distributing these replicas across the network in a transparent manner is the domain of network transparency.

1.4.1.4 Fragmentation Transparency

The final form of transparency that needs to be addressed within the context of a distributed database system is that of fragmentation transparency. In Chapter 3 we discuss and justify the fact that it is commonly desirable to divide each database relation into smaller fragments and treat each fragment as a separate database object (i.e., another relation). This is commonly done for reasons of performance, availability, and reliability. Furthermore, fragmentation can reduce the negative effects of replication. Each replica is not the full relation but only a subset of it; thus less space is required and fewer data items need be managed.

There are two general types of fragmentation alternatives. In one case, called *horizontal fragmentation*, a relation is partitioned into a set of sub-relations each of which have a subset of the tuples (rows) of the original relation. The second alternative is *vertical fragmentation* where each sub-relation is defined on a subset of the attributes (columns) of the original relation.

When database objects are fragmented, we have to deal with the problem of handling user queries that are specified on entire relations but have to be executed on subrelations. In other words, the issue is one of finding a query processing strategy based on the fragments rather than the relations, even though the queries are specified on the latter. Typically, this requires a translation from what is called a *global query* to several *fragment queries*. Since the fundamental issue of dealing with fragmentation transparency is one of query processing, we defer the discussion of techniques by which this translation can be performed until Chapter 7.

1.4.1.5 Who Should Provide Transparency?

In previous sections we discussed various possible forms of transparency within a distributed computing environment. Obviously, to provide easy and efficient access by novice users to the services of the DBMS, one would want to have full transparency, involving all the various types that we discussed. Nevertheless, the level of transparency is inevitably a compromise between ease of use and the difficulty and overhead cost of providing high levels of transparency. For example, Gray argues that full transparency makes the management of distributed data very difficult and claims that "applications coded with transparent access to geographically distributed databases have: poor manageability, poor modularity, and poor message performance" [Gray, 1989]. He proposes a remote procedure call mechanism between the requestor users and the server DBMSs whereby the users would direct their queries to a specific DBMS. This is indeed the approach commonly taken by client/server systems that we discuss shortly.

What has not yet been discussed is who is responsible for providing these services. It is possible to identify three distinct layers at which the transparency services can be provided. It is quite common to treat these as mutually exclusive means of providing the service, although it is more appropriate to view them as complementary.

We could leave the responsibility of providing transparent access to data resources to the access layer. The transparency features can be built into the user language, which then translates the requested services into required operations. In other words, the compiler or the interpreter takes over the task and no transparent service is provided to the implementer of the compiler or the interpreter.

The second layer at which transparency can be provided is the operating system level. State-of-the-art operating systems provide some level of transparency to system users. For example, the device drivers within the operating system handle the details of getting each piece of peripheral equipment to do what is requested. The typical computer user, or even an application programmer, does not normally write device drivers to interact with individual peripheral equipment; that operation is transparent to the user.

Providing transparent access to resources at the operating system level can obviously be extended to the distributed environment, where the management of the network resource is taken over by the distributed operating system or the middleware if the distributed DBMS is implemented over one. There are two potential problems with this approach. The first is that not all commercially available distributed operating systems provide a reasonable level of transparency in network management. The second problem is that some applications do not wish to be shielded from the details of distribution and need to access them for specific performance tuning.

The third layer at which transparency can be supported is within the DBMS. The transparency and support for database functions provided to the DBMS designers by an underlying operating system is generally minimal and typically limited to very fundamental operations for performing certain tasks. It is the responsibility of the DBMS to make all the necessary translations from the operating system to the higher-level user interface. This mode of operation is the most common method today. There are, however, various problems associated with leaving the task of providing full transparency to the DBMS. These have to do with the interaction of the operating system with the distributed DBMS and are discussed throughout this book.

A hierarchy of these transparencies is shown in Figure 1.6. It is not always easy to delineate clearly the levels of transparency, but such a figure serves an important instructional purpose even if it is not fully correct. To complete the picture we have added a "language transparency" layer, although it is not discussed in this chapter. With this generic layer, users have high-level access to the data (e.g., fourth-generation languages, graphical user interfaces, natural language access).

1.4.2 Reliability Through Distributed Transactions

Distributed DBMSs are intended to improve reliability since they have replicated components and, thereby eliminate single points of failure. The failure of a single site, or the failure of a communication link which makes one or more sites unreachable, is not sufficient to bring down the entire system. In the case of a distributed database, this means that some of the data may be unreachable, but with proper care, users

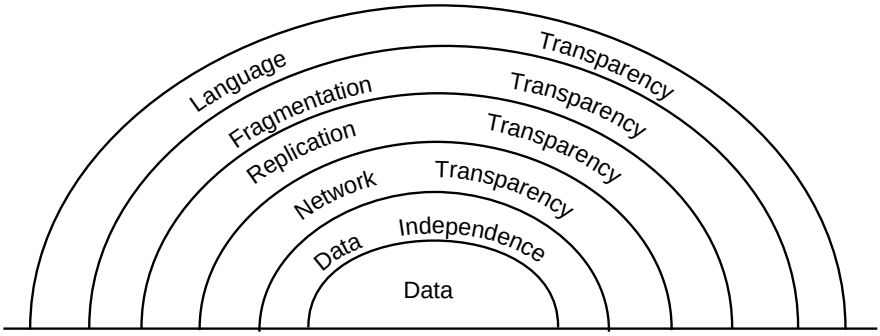

Fig. 1.6 Layers of Transparency

may be permitted to access other parts of the distributed database. The "proper care" comes in the form of support for distributed transactions and application protocols.

We discuss transactions and transaction processing in detail in Chapters 10–12. A *transaction* is a basic unit of consistent and reliable computing, consisting of a sequence of database operations executed as an atomic action. It transforms a consistent database state to another consistent database state even when a number of such transactions are executed concurrently (sometimes called *concurrency transparency*), and even when failures occur (also called *failure atomicity*). Therefore, a DBMS that provides full transaction support guarantees that concurrent execution of user transactions will not violate database consistency in the face of system failures as long as each transaction is correct, i.e., obeys the integrity rules specified on the database.

Let us give an example of a transaction based on the engineering firm example that we introduced earlier. Assume that there is an application that updates the salaries of all the employees by 10%. It is desirable to encapsulate the query (or the program code) that accomplishes this task within transaction boundaries. For example, if a system failure occurs half-way through the execution of this program, we would like the DBMS to be able to determine, upon recovery, where it left off and continue with its operation (or start all over again). This is the topic of failure atomicity. Alternatively, if some other user runs a query calculating the average salaries of the employees in this firm while the original update action is going on, the calculated result will be in error. Therefore we would like the system to be able to synchronize the *concurrent* execution of these two programs. To encapsulate a query (or a program code) within transactional boundaries, it is sufficient to declare the begin of the transaction and its end:

```
Begin_transaction SALARY_UPDATE
begin
  EXEC SQL UPDATE   PAY
           SET      SAL = SAL*1.1
end.
```

Distributed transactions execute at a number of sites at which they access the local database. The above transaction, for example, will execute in Boston, Waterloo, Paris and San Francisco since the data are distributed at these sites. With full support for distributed transactions, user applications can access a single logical image of the database and rely on the distributed DBMS to ensure that their requests will be executed correctly no matter what happens in the system. "Correctly" means that user applications do not need to be concerned with coordinating their accesses to individual local databases nor do they need to worry about the possibility of site or communication link failures during the execution of their transactions. This illustrates the link between distributed transactions and transparency, since both involve issues related to distributed naming and directory management, among other things.

Providing transaction support requires the implementation of distributed concurrency control (Chapter 11) and distributed reliability (Chapter 12) protocols — in particular, two-phase commit (2PC) and distributed recovery protocols — which are significantly more complicated than their centralized counterparts. Supporting replicas requires the implementation of replica control protocols that enforce a specified semantics of accessing them (Chapter 13).

1.4.3 Improved Performance

The case for the improved performance of distributed DBMSs is typically made based on two points. First, a distributed DBMS fragments the conceptual database, enabling data to be stored in close proximity to its points of use (also called *data localization*). This has two potential advantages:

1. Since each site handles only a portion of the database, contention for CPU and I/O services is not as severe as for centralized databases.
2. Localization reduces remote access delays that are usually involved in wide area networks (for example, the minimum round-trip message propagation delay in satellite-based systems is about 1 second).

Most distributed DBMSs are structured to gain maximum benefit from data localization. Full benefits of reduced contention and reduced communication overhead can be obtained only by a proper fragmentation and distribution of the database.

This point relates to the overhead of distributed computing if the data have to reside at remote sites and one has to access it by remote communication. The argument is that it is better, in these circumstances, to distribute the data management functionality to where the data are located rather than moving large amounts of data. This has lately become a topic of contention. Some argue that with the widespread use of high-speed, high-capacity networks, distributing data and data management functions no longer makes sense and that it may be much simpler to store data at a central site and access it (by downloading) over high-speed networks. This argument, while appealing, misses the point of distributed databases. First of all, in

most of today's applications, data are distributed; what may be open for debate is how and where we process it. Second, and more important, point is that this argument does not distinguish between bandwidth (the capacity of the computer links) and latency (how long it takes for data to be transmitted). Latency is inherent in the distributed environments and there are physical limits to how fast we can send data over computer networks. As indicated above, for example, satellite links take about half-a-second to transmit data between two ground stations. This is a function of the distance of the satellites from the earth and there is nothing that we can do to improve that performance. For some applications, this might constitute an unacceptable delay.

The second case point is that the inherent parallelism of distributed systems may be exploited for inter-query and intra-query parallelism. Inter-query parallelism results from the ability to execute multiple queries at the same time while intra-query parallelism is achieved by breaking up a single query into a number of subqueries each of which is executed at a different site, accessing a different part of the distributed database.

If the user access to the distributed database consisted only of querying (i.e., read-only access), then provision of inter-query and intra-query parallelism would imply that as much of the database as possible should be replicated. However, since most database accesses are not read-only, the mixing of read and update operations requires the implementation of elaborate concurrency control and commit protocols.

1.4.4 Easier System Expansion

In a distributed environment, it is much easier to accommodate increasing database sizes. Major system overhauls are seldom necessary; expansion can usually be handled by adding processing and storage power to the network. Obviously, it may not be possible to obtain a linear increase in "power," since this also depends on the overhead of distribution. However, significant improvements are still possible.

One aspect of easier system expansion is economics. It normally costs much less to put together a system of "smaller" computers with the equivalent power of a single big machine. In earlier times, it was commonly believed that it would be possible to purchase a fourfold powerful computer if one spent twice as much. This was known as Grosh's law. With the advent of microcomputers and workstations, and their price/performance characteristics, this law is considered invalid.

This should not be interpreted to mean that mainframes are dead; this is not the point that we are making here. Indeed, in recent years, we have observed a resurgence in the world-wide sale of mainframes. The point is that for many applications, it is more economical to put together a distributed computer system (whether composed of mainframes or workstations) with sufficient power than it is to establish a single, centralized system to run these tasks. In fact, the latter may not even be feasible these days.

1.5 Complications Introduced by Distribution

The problems encountered in database systems take on additional complexity in a distributed environment, even though the basic underlying principles are the same. Furthermore, this additional complexity gives rise to new problems influenced mainly by three factors.

First, data may be replicated in a distributed environment. A distributed database can be designed so that the entire database, or portions of it, reside at different sites of a computer network. It is not essential that every site on the network contain the database; it is only essential that there be more than one site where the database resides. The possible duplication of data items is mainly due to reliability and efficiency considerations. Consequently, the distributed database system is responsible for (1) choosing one of the stored copies of the requested data for access in case of retrievals, and (2) making sure that the effect of an update is reflected on each and every copy of that data item.

Second, if some sites fail (e.g., by either hardware or software malfunction), or if some communication links fail (making some of the sites unreachable) while an update is being executed, the system must make sure that the effects will be reflected on the data residing at the failing or unreachable sites as soon as the system can recover from the failure.

The third point is that since each site cannot have instantaneous information on the actions currently being carried out at the other sites, the synchronization of transactions on multiple sites is considerably harder than for a centralized system.

These difficulties point to a number of potential problems with distributed DBMSs. These are the inherent complexity of building distributed applications, increased cost of replicating resources, and, more importantly, managing distribution, the devolution of control to many centers and the difficulty of reaching agreements, and the exacerbated security concerns (the secure communication channel problem). These are well-known problems in distributed systems in general, and, in this book, we discuss their manifestations within the context of distributed DBMS and how they can be addressed.

1.6 Design Issues

In Section 1.4, we discussed the promises of distributed DBMS technology, highlighting the challenges that need to be overcome in order to realize them. In this section we build on this discussion by presenting the design issues that arise in building a distributed DBMS. These issues will occupy much of the remainder of this book.

1.6.1 Distributed Database Design

The question that is being addressed is how the database and the applications that run against it should be placed across the sites. There are two basic alternatives to placing data: *partitioned* (or *non-replicated*) and *replicated*. In the partitioned scheme the database is divided into a number of disjoint partitions each of which is placed at a different site. Replicated designs can be either *fully replicated* (also called *fully duplicated*) where the entire database is stored at each site, or *partially replicated* (or *partially duplicated*) where each partition of the database is stored at more than one site, but not at all the sites. The two fundamental design issues are *fragmentation*, the separation of the database into partitions called *fragments*, and *distribution*, the optimum distribution of fragments.

The research in this area mostly involves mathematical programming in order to minimize the combined cost of storing the database, processing transactions against it, and message communication among sites. The general problem is NP-hard. Therefore, the proposed solutions are based on heuristics. Distributed database design is the topic of Chapter 3.

1.6.2 Distributed Directory Management

A directory contains information (such as descriptions and locations) about data items in the database. Problems related to directory management are similar in nature to the database placement problem discussed in the preceding section. A directory may be global to the entire DDBS or local to each site; it can be centralized at one site or distributed over several sites; there can be a single copy or multiple copies. We briefly discuss these issues in Chapter 3.

1.6.3 Distributed Query Processing

Query processing deals with designing algorithms that analyze queries and convert them into a series of data manipulation operations. The problem is how to decide on a strategy for executing each query over the network in the most cost-effective way, however cost is defined. The factors to be considered are the distribution of data, communication costs, and lack of sufficient locally-available information. The objective is to optimize where the inherent parallelism is used to improve the performance of executing the transaction, subject to the above-mentioned constraints. The problem is NP-hard in nature, and the approaches are usually heuristic. Distributed query processing is discussed in detail in Chapter 6 - 8.

1.6.4 Distributed Concurrency Control

Concurrency control involves the synchronization of accesses to the distributed database, such that the integrity of the database is maintained. It is, without any doubt, one of the most extensively studied problems in the DDBS field. The concurrency control problem in a distributed context is somewhat different than in a centralized framework. One not only has to worry about the integrity of a single database, but also about the consistency of multiple copies of the database. The condition that requires all the values of multiple copies of every data item to converge to the same value is called *mutual consistency*.

The alternative solutions are too numerous to discuss here, so we examine them in detail in Chapter 11. Let us only mention that the two general classes are *pessimistic* , synchronizing the execution of user requests before the execution starts, and *optimistic*, executing the requests and then checking if the execution has compromised the consistency of the database. Two fundamental primitives that can be used with both approaches are *locking*, which is based on the mutual exclusion of accesses to data items, and *timestamping*, where the transaction executions are ordered based on timestamps. There are variations of these schemes as well as hybrid algorithms that attempt to combine the two basic mechanisms.

1.6.5 Distributed Deadlock Management

The deadlock problem in DDBSs is similar in nature to that encountered in operating systems. The competition among users for access to a set of resources (data, in this case) can result in a deadlock if the synchronization mechanism is based on locking. The well-known alternatives of prevention, avoidance, and detection/recovery also apply to DDBSs. Deadlock management is covered in Chapter 11.

1.6.6 Reliability of Distributed DBMS

We mentioned earlier that one of the potential advantages of distributed systems is improved reliability and availability. This, however, is not a feature that comes automatically. It is important that mechanisms be provided to ensure the consistency of the database as well as to detect failures and recover from them. The implication for DDBSs is that when a failure occurs and various sites become either inoperable or inaccessible, the databases at the operational sites remain consistent and up to date. Furthermore, when the computer system or network recovers from the failure, the DDBSs should be able to recover and bring the databases at the failed sites up-to-date. This may be especially difficult in the case of network partitioning, where the sites are divided into two or more groups with no communication among them. Distributed reliability protocols are the topic of Chapter 12.

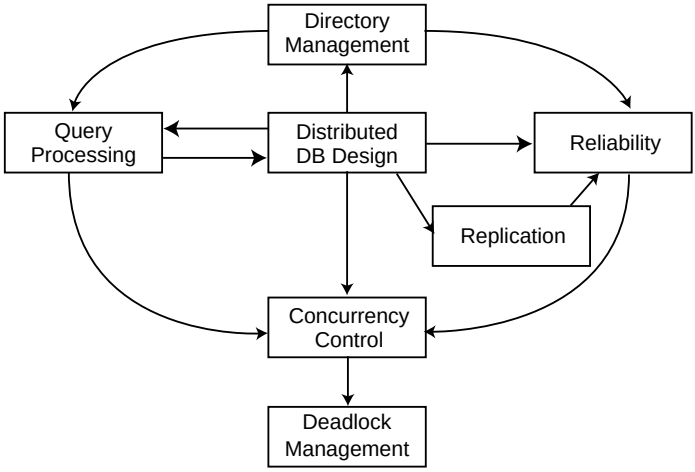

Fig. 1.7 Relationship Among Research Issues

1.6.7 Replication

If the distributed database is (partially or fully) replicated, it is necessary to implement protocols that ensure the consistency of the replicas,i.e., copies of the same data item have the same value. These protocols can be *eager* in that they force the updates to be applied to all the replicas before the transaction completes, or they may be *lazy* so that the transaction updates one copy (called the *master*) from which updates are propagated to the others after the transaction completes. We discuss replication protocols in Chapter 13.

1.6.8 Relationship among Problems

Naturally, these problems are not isolated from one another. Each problem is affected by the solutions found for the others, and in turn affects the set of feasible solutions for them. In this section we discuss how they are related.

The relationship among the components is shown in Figure 1.7. The design of distributed databases affects many areas. It affects directory management, because the definition of fragments and their placement determine the contents of the directory (or directories) as well as the strategies that may be employed to manage them. The same information (i.e., fragment structure and placement) is used by the query processor to determine the query evaluation strategy. On the other hand, the access and usage patterns that are determined by the query processor are used as inputs to the data distribution and fragmentation algorithms. Similarly, directory placement and contents influence the processing of queries.

The replication of fragments when they are distributed affects the concurrency control strategies that might be employed. As we will study in Chapter 11, some concurrency control algorithms cannot be easily used with replicated databases. Similarly, usage and access patterns to the database will influence the concurrency control algorithms. If the environment is update intensive, the necessary precautions are quite different from those in a query-only environment.

There is a strong relationship among the concurrency control problem, the deadlock management problem, and reliability issues. This is to be expected, since together they are usually called the *transaction management* problem. The concurrency control algorithm that is employed will determine whether or not a separate deadlock management facility is required. If a locking-based algorithm is used, deadlocks will occur, whereas they will not if timestamping is the chosen alternative.

Reliability mechanisms involve both local recovery techniques and distributed reliability protocols. In that sense, they both influence the choice of the concurrency control techniques and are built on top of them. Techniques to provide reliability also make use of data placement information since the existence of duplicate copies of the data serve as a safeguard to maintain reliable operation.

Finally, the need for replication protocols arise if data distribution involves replicas. As indicated above, there is a strong relationship between replication protocols and concurrency control techniques, since both deal with the consistency of data, but from different perspectives. Furthermore, the replication protocols influence distributed reliability techniques such as commit protocols. In fact, it is sometimes suggested (wrongly, in our view) that replication protocols can be used instead of implementing distributed commit protocols.

1.6.9 Additional Issues

The above design issues cover what may be called "traditional" distributed database systems. The environment has changed significantly since these topics started to be investigated, posing additional challenges and opportunities.

One of the important developments has been the move towards "looser" federation among data sources, which may also be heterogeneous. As we discuss in the next section, this has given rise to the development of multidatabase systems (also called *federated databases* and *data integration systems*) that require re-investigation of some of the fundamental database techniques. These systems constitute an important part of today's distributed environment. We discuss database design issues in multidatabase systems (i.e., *database integration*) in Chapter 4 and the query processing challenges in Chapter 9.

The growth of the Internet as a fundamental networking platform has raised important questions about the assumptions underlying distributed database systems. Two issues are of particular concern to us. One is the re-emergence of peer-to-peer computing, and the other is the development and growth of the World Wide Web (web for short). Both of these aim at improving data sharing, but take different

approaches and pose different data management challenges. We discuss peer-to-peer data management in Chapter 16 and web data management in Chapter 17.

We should note that peer-to-peer is not a new concept in distributed databases, as we discuss in the next section. However, their new re-incarnation has significant differences from the earlier versions. In Chapter 16, it is these new versions that we focus on.

Finally, as earlier noted, there is a strong relationship between distributed databases and parallel databases. Although the former assumes each site to be a single logical computer, most of these installations are, in fact, parallel clusters. Thus, while most of the book focuses on issues that arise in managing data distributed across these sites, interesting data management issues exist within a single logical site that may be a parallel system. We discuss these issues in Chapter 14.

1.7 Distributed DBMS Architecture

The architecture of a system defines its structure. This means that the components of the system are identified, the function of each component is specified, and the interrelationships and interactions among these components are defined. The specification of the architecture of a system requires identification of the various modules, with their interfaces and interrelationships, in terms of the data and control flow through the system.

In this section we develop three "reference" architectures[2] for a distributed DBMS: client/server systems, peer-to-peer distributed DBMS, and multidatabase systems. These are "idealized" views of a DBMS in that many of the commercially available systems may deviate from them; however, the architectures will serve as a reasonable framework within which the issues related to distributed DBMS can be discussed.

We first start with a brief presentation of the "ANSI/SPARC architecture", which is a *datalogical* approach to defining a DBMS architecture – it focuses on the different user classes and roles and their varying views on data. This architecture is helpful in putting certain concepts we have discussed so far in their proper perspective. We then have a short discussion of a generic architecture of a centralized DBMSs, that we subsequently extend to identify the set of alternative architectures for a distributed DBMS. Whithin this characterization, we focus on the three alternatives that we identified above.

1.7.1 ANSI/SPARC Architecture

In late 1972, the Computer and Information Processing Committee (X3) of the American National Standards Institute (ANSI) established a Study Group on Database

[2] A reference architecture is commonly created by standards developers to clearly define the interfaces that need to be standardized.

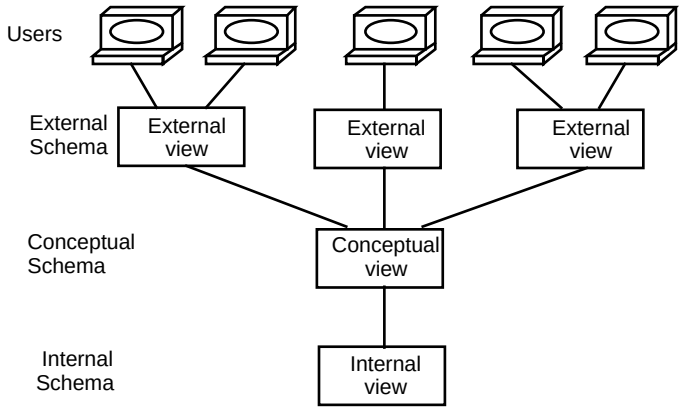

Fig. 1.8 The ANSI/SPARC Architecture

Management Systems under the auspices of its Standards Planning and Requirements Committee (SPARC). The mission of the study group was to study the *feasibility* of setting up standards in this area, as well as determining which aspects should be standardized if it was feasible. The study group issued its interim report in 1975 [ANSI/SPARC, 1975], and its final report in 1977 [Tsichritzis and Klug, 1978]. The architectural framework proposed in these reports came to be known as the "ANSI/SPARC architecture," its full title being "ANSI/X3/SPARC DBMS Framework." The study group proposed that the interfaces be standardized, and defined an architectural framework that contained 43 interfaces, 14 of which would deal with the physical storage subsystem of the computer and therefore not be considered essential parts of the DBMS architecture.

A simplified version of the ANSI/SPARC architecture is depicted in Figure 1.8. There are three views of data: the *external view*, which is that of the end user, who might be a programmer; the *internal view*, that of the system or machine; and the *conceptual view*, that of the enterprise. For each of these views, an appropriate schema definition is required.

At the lowest level of the architecture is the internal view, which deals with the physical definition and organization of data. The location of data on different storage devices and the access mechanisms used to reach and manipulate data are the issues dealt with at this level. At the other extreme is the external view, which is concerned with how users view the database. An individual user's view represents the portion of the database that will be accessed by that user as well as the relationships that the user would like to see among the data. A view can be shared among a number of users, with the collection of user views making up the external schema. In between these two ends is the conceptual schema, which is an abstract definition of the database. It is the "real world" view of the enterprise being modeled in the database [Yormark, 1977]. As such, it is supposed to represent the data and the relationships among data without considering the requirements of individual applications or the restrictions of the physical storage media. In reality, however, it is not possible to ignore these

requirements completely, due to performance reasons. The transformation between these three levels is accomplished by mappings that specify how a definition at one level can be obtained from a definition at another level.

This perspective is important, because it provides the basis for data independence that we discussed earlier. The separation of the external schemas from the conceptual schema enables *logical data independence*, while the separation of the conceptual schema from the internal schema allows *physical data independence*.

1.7.2 A Generic Centralized DBMS Architecture

A DBMS is a reentrant program shared by multiple processes (*transactions*), that run database programs. When running on a general purpose computer, a DBMS is interfaced with two other components: the communication subsystem and the operating system. The communication subsystem permits interfacing the DBMS with other subsystems in order to communicate with applications. For example, the terminal monitor needs to communicate with the DBMS to run interactive transactions. The operating system provides the interface between the DBMS and computer resources (processor, memory, disk drives, etc.).

The functions performed by a DBMS can be layered as in Figure 1.9, where the arrows indicate the direction of the data and the control flow. Taking a top-down approach, the layers are the interface, control, compilation, execution, data access, and consistency management.

The *interface layer* manages the interface to the applications. There can be several interfaces such as, in the case of relational DBMSs discussed in Chapter 2, SQL embedded in a host language, such as C and QBE (Query-by-Example). Database application programs are executed against external *views* of the database. For an application, a view is useful in representing its particular perception of the database (shared by many applications). A view in relational DBMSs is a virtual relation derived from base relations by applying relational algebra operations.[3] These concepts are defined more precisely in Chapter 2, but they are usually covered in undergraduate database courses, so we expect many readers to be familiar with them. View management consists of translating the user query from external data to conceptual data.

The *control layer* controls the query by adding semantic integrity predicates and authorization predicates. Semantic integrity constraints and authorizations are usually specified in a declarative language, as discussed in Chapter 5. The output of this layer is an enriched query in the high-level language accepted by the interface.

The *query processing* (or *compilation*) layer maps the query into an optimized sequence of lower-level operations. This layer is concerned with performance. It

[3] Note that this does not mean that the real-world views are, or should be, specified in relational algebra. On the contrary, they are specified by some high-level data language such as SQL. The translation from one of these languages to relational algebra is now well understood, and the effects of the view definition can be specified in terms of relational algebra operations.

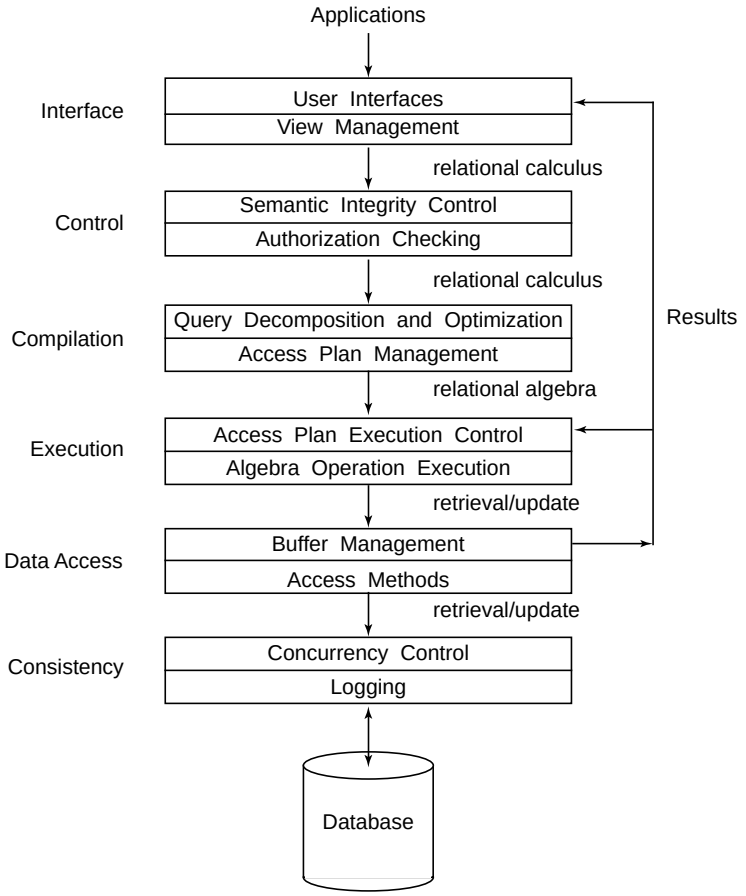

Fig. 1.9 Functional Layers of a Centralized DBMS

decomposes the query into a tree of algebra operations and tries to find the "optimal" ordering of the operations. The result is stored in an access plan. The output of this layer is a query expressed in lower-level code (algebra operations).

The *execution layer* directs the execution of the access plans, including transaction management (commit, restart) and synchronization of algebra operations. It interprets the relational operations by calling the data access layer through the retrieval and update requests.

The *data access layer* manages the data structures that implement the files, indices, etc. It also manages the buffers by caching the most frequently accessed data. Careful use of this layer minimizes the access to disks to get or write data.

Finally, the *consistency layer* manages concurrency control and logging for update requests. This layer allows transaction, system, and media recovery after failure.

1.7.3 Architectural Models for Distributed DBMSs

We now consider the possible ways in which a distributed DBMS may be architected. We use a classification (Figure 1.10) that organizes the systems as characterized with respect to (1) the autonomy of local systems, (2) their distribution, and (3) their heterogeneity.

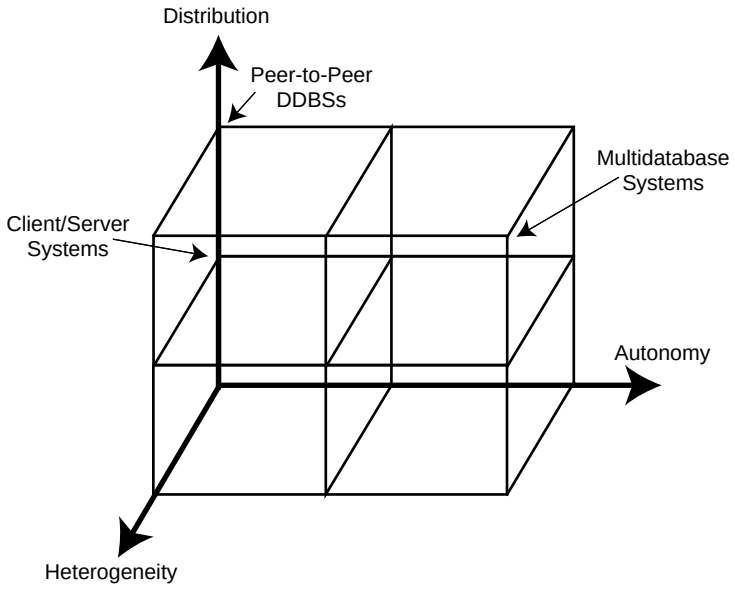

Fig. 1.10 DBMS Implementation Alternatives

1.7.4 Autonomy

Autonomy, in this context, refers to the distribution of control, not of data. It indicates the degree to which individual DBMSs can operate independently. Autonomy is a function of a number of factors such as whether the component systems (i.e., individual DBMSs) exchange information, whether they can independently execute transactions, and whether one is allowed to modify them. Requirements of an autonomous system have been specified as follows [Gligor and Popescu-Zeletin, 1986]:

1. The local operations of the individual DBMSs are not affected by their participation in the distributed system.

2. The manner in which the individual DBMSs process queries and optimize them should not be affected by the execution of global queries that access multiple databases.
3. System consistency or operation should not be compromised when individual DBMSs join or leave the distributed system.

On the other hand, the dimensions of autonomy can be specified as follows [Du and Elmagarmid, 1989]:

1. Design autonomy: Individual DBMSs are free to use the data models and transaction management techniques that they prefer.
2. Communication autonomy: Each of the individual DBMSs is free to make its own decision as to what type of information it wants to provide to the other DBMSs or to the software that controls their global execution.
3. Execution autonomy: Each DBMS can execute the transactions that are submitted to it in any way that it wants to.

We will use a classification that covers the important aspects of these features. One alternative is *tight integration*, where a single-image of the entire database is available to any user who wants to share the information, which may reside in multiple databases. From the users' perspective, the data are logically integrated in one database. In these tightly-integrated systems, the data managers are implemented so that one of them is in control of the processing of each user request even if that request is serviced by more than one data manager. The data managers do not typically operate as independent DBMSs even though they usually have the functionality to do so.

Next we identify *semiautonomous* systems that consist of DBMSs that can (and usually do) operate independently, but have decided to participate in a federation to make their local data sharable. Each of these DBMSs determine what parts of their own database they will make accessible to users of other DBMSs. They are not fully autonomous systems because they need to be modified to enable them to exchange information with one another.

The last alternative that we consider is *total isolation*, where the individual systems are stand-alone DBMSs that know neither of the existence of other DBMSs nor how to communicate with them. In such systems, the processing of user transactions that access multiple databases is especially difficult since there is no global control over the execution of individual DBMSs.

It is important to note at this point that the three alternatives that we consider for autonomous systems are not the only possibilities. We simply highlight the three most popular ones.

1.7.5 Distribution

Whereas autonomy refers to the distribution (or decentralization) of control, the distribution dimension of the taxonomy deals with data. Of course, we are considering the physical distribution of data over multiple sites; as we discussed earlier, the user sees the data as one logical pool. There are a number of ways DBMSs have been distributed. We abstract these alternatives into two classes: *client/server* distribution and *peer-to-peer* distribution (or *full* distribution). Together with the non-distributed option, the taxonomy identifies three alternative architectures.

The client/server distribution concentrates data management duties at servers while the clients focus on providing the application environment including the user interface. The communication duties are shared between the client machines and servers. Client/server DBMSs represent a practical compromise to distributing functionality. There are a variety of ways of structuring them, each providing a different level of distribution. With respect to the framework, we abstract these differences and leave that discussion to Section 1.7.8, which we devote to client/server DBMS architectures. What is important at this point is that the sites on a network are distinguished as "clients" and "servers" and their functionality is different.

In *peer-to-peer systems*, there is no distinction of client machines versus servers. Each machine has full DBMS functionality and can communicate with other machines to execute queries and transactions. Most of the very early work on distributed database systems have assumed peer-to-peer architecture. Therefore, our main focus in this book are on peer-to-peer systems (also called *fully distributed*), even though many of the techniques carry over to client/server systems as well.

1.7.6 Heterogeneity

Heterogeneity may occur in various forms in distributed systems, ranging from hardware heterogeneity and differences in networking protocols to variations in data managers. The important ones from the perspective of this book relate to data models, query languages, and transaction management protocols. Representing data with different modeling tools creates heterogeneity because of the inherent expressive powers and limitations of individual data models. Heterogeneity in query languages not only involves the use of completely different data access paradigms in different data models (set-at-a-time access in relational systems versus record-at-a-time access in some object-oriented systems), but also covers differences in languages even when the individual systems use the same data model. Although SQL is now the standard relational query language, there are many different implementations and every vendor's language has a slightly different flavor (sometimes even different semantics, producing different results).

1.7.7 Architectural Alternatives

The distribution of databases, their possible heterogeneity, and their autonomy are orthogonal issues. Consequently, following the above characterization, there are 18 different possible architectures. Not all of these architectural alternatives that form the design space are meaningful. Furthermore, not all are relevant from the perspective of this book.

In Figure 1.10, we have identified three alternative architectures that are the focus of this book and that we discuss in more detail in the next three subsections: (A0, D1, H0) that corresponds to client/server distributed DBMSs, (A0, D2, H0) that is a peer-to-peer distributed DBMS and (A2, D2, H1) which represents a (peer-to-peer) distributed, heterogeneous multidatabase system. Note that we discuss the heterogeneity issues within the context of one system architecture, although the issue arises in other models as well.

1.7.8 Client/Server Systems

Client/server DBMSs entered the computing scene at the beginning of 1990's and have made a significant impact on both the DBMS technology and the way we do computing. The general idea is very simple and elegant: distinguish the functionality that needs to be provided and divide these functions into two classes: server functions and client functions. This provides a *two-level architecture* which makes it easier to manage the complexity of modern DBMSs and the complexity of distribution.

As with any highly popular term, client/server has been much abused and has come to mean different things. If one takes a process-centric view, then any process that requests the services of another process is its client and vice versa. However, it is important to note that "client/server computing" and "client/server DBMS," as it is used in our context, do not refer to processes, but to actual machines. Thus, we focus on what software should run on the client machines and what software should run on the server machine.

Put this way, the issue is clearer and we can begin to study the differences in client and server functionality. The functionality allocation between clients and serves differ in different types of distributed DBMSs (e.g., relational versus object-oriented). In relational systems, the server does most of the data management work. This means that all of query processing and optimization, transaction management and storage management is done at the server. The client, in addition to the application and the user interface, has a *DBMS client* module that is responsible for managing the data that is cached to the client and (sometimes) managing the transaction locks that may have been cached as well. It is also possible to place consistency checking of user queries at the client side, but this is not common since it requires the replication of the system catalog at the client machines. Of course, there is operating system and communication software that runs on both the client and the server, but we only focus on the DBMS related functionality. This architecture, depicted in Figure 1.11,

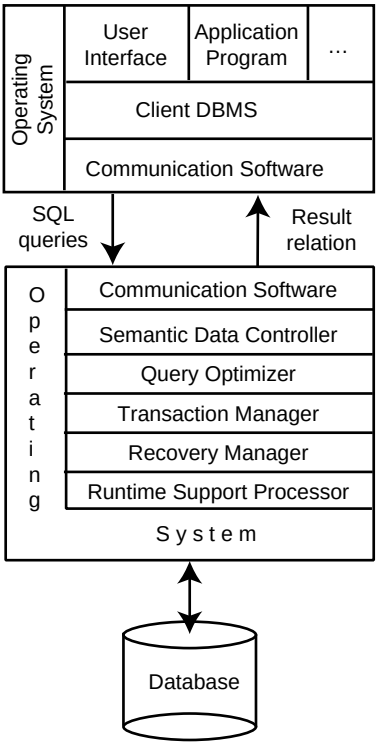

Fig. 1.11 Client/Server Reference Architecture

is quite common in relational systems where the communication between the clients and the server(s) is at the level of SQL statements. In other words, the client passes SQL queries to the server without trying to understand or optimize them. The server does most of the work and returns the result relation to the client.

There are a number of different types of client/server architecture. The simplest is the case where there is only one server which is accessed by multiple clients. We call this *multiple client/single server*. From a data management perspective, this is not much different from centralized databases since the database is stored on only one machine (the server) that also hosts the software to manage it. However, there are some (important) differences from centralized systems in the way transactions are executed and caches are managed. We do not consider such issues at this point. A more sophisticated client/server architecture is one where there are multiple servers in the system (the so-called *multiple client/multiple server* approach). In this case, two alternative management strategies are possible: either each client manages its own connection to the appropriate server or each client knows of only its "home server" which then communicates with other servers as required. The former approach simplifies server code, but loads the client machines with additional responsibilities. This leads to what has been called "heavy client" systems. The latter approach, on

the other hand, concentrates the data management functionality at the servers. Thus, the transparency of data access is provided at the server interface, leading to "light clients."

From a datalogical perspective, client/server DBMSs provide the same view of data as do peer-to-peer systems that we discuss next. That is, they give the user the appearance of a logically single database, while at the physical level data **may** be distributed. Thus the primary distinction between client/server systems and peer-to-peer ones is not in the level of transparency that is provided to the users and applications, but in the architectural paradigm that is used to realize this level of transparency.

Client/server can be naturally extended to provide for a more efficient function distribution on different kinds of servers: *client servers* run the user interface (e.g., web servers), *application servers* run application programs, and *database servers* run database management functions. This leads to the present trend in three-tier distributed system architecture, where sites are organized as specialized servers rather than as general-purpose computers.

The original idea, which is to offload the database management functions to a special server, dates back to the early 1970s [Canaday et al., 1974]. At the time, the computer on which the database system was run was called the *database machine*, *database computer*, or *backend computer*, while the computer that ran the applications was called the *host computer*. More recent terms for these are the *database server* and *application server*, respectively. Figure 1.12 illustrates a simple view of the database server approach, with application servers connected to one database server via a communication network.

The database server approach, as an extension of the classical client/server architecture, has several potential advantages. First, the single focus on data management makes possible the development of specific techniques for increasing data reliability and availability, e.g. using parallelism. Second, the overall performance of database management can be significantly enhanced by the tight integration of the database system and a dedicated database operating system. Finally, a database server can also exploit recent hardware architectures, such as multiprocessors or clusters of PC servers to enhance both performance and data availability.

Although these advantages are significant, they can be offset by the overhead introduced by the additional communication between the application and the data servers. This is an issue, of course, in classical client/server systems as well, but in this case there is an additional layer of communication to worry about. The communication cost can be amortized only if the server interface is sufficiently high level to allow the expression of complex queries involving intensive data processing.

The application server approach (indeed, a n-tier distributed approach) can be extended by the introduction of multiple database servers and multiple application servers (Figure 1.13), as can be done in classical client/server architectures. In this case, it is typically the case that each application server is dedicated to one or a few applications, while database servers operate in the multiple server fashion discussed above.

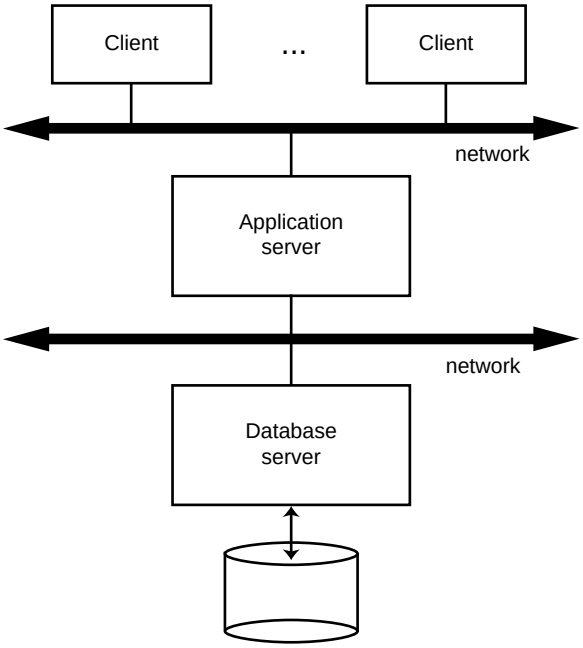

Fig. 1.12 Database Server Approach

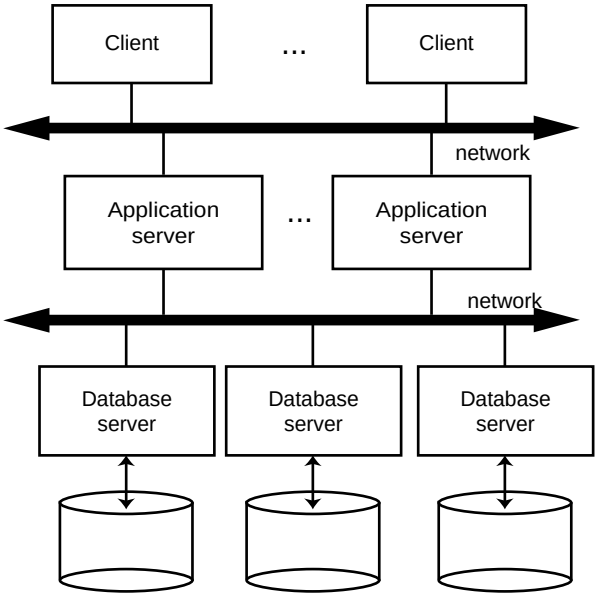

Fig. 1.13 Distributed Database Servers

1.7.9 Peer-to-Peer Systems

If the term "client/server" is loaded with different interpretations, "peer-to-peer" is even worse as its meaning has changed and evolved over the years. As noted earlier, the early works on distributed DBMSs all focused on peer-to-peer architectures where there was no differentiation between the functionality of each site in the system[4]. After a decade of popularity of client/server computing, peer-to-peer have made a comeback in the last few years (primarily spurred by file sharing applications) and some have even positioned peer-to-peer data management as an alternative to distributed DBMSs. While this may be a stretch, modern peer-to-peer systems have two important differences from their earlier relatives. The first is the massive distribution in current systems. While in the early days we focused on a few (perhaps at most tens of) sites, current systems consider thousands of sites. The second is the inherent heterogeneity of every aspect of the sites and their autonomy. While this has always been a concern of distributed databases, as discussed earlier, coupled with massive distribution, site heterogeneity and autonomy take on an added significance, disallowing some of the approaches from consideration.

Discussing peer-to-peer database systems within this backdrop poses real challenges; the unique issues of database management over the "modern" peer-to-peer architectures are still being investigated. What we choose to do, in this book, is to initially focus on the classical meaning of peer-to-peer (the same functionality of each site), since the principles and fundamental techniques of these systems are very similar to those of client/server systems, and discuss the modern peer-to-peer database issues in a separate chapter (Chapter 16).

Let us start the description of the architecture by looking at the data organizational view. We first note that the physical data organization on each machine may be, and probably is, different. This means that there needs to be an individual internal schema definition at each site, which we call the *local internal schema* (LIS). The enterprise view of the data is described by the *global conceptual schema* (GCS), which is global because it describes the logical structure of the data at all the sites.

To handle data fragmentation and replication, the logical organization of data at each site needs to be described. Therefore, there needs to be a third layer in the architecture, the *local conceptual schema* (LCS). In the architectural model we have chosen, then, the global conceptual schema is the union of the local conceptual schemas. Finally, user applications and user access to the database is supported by *external schemas* (ESs), defined as being above the global conceptual schema.

This architecture model, depicted in Figure 1.14, provides the levels of transparency discussed earlier. Data independence is supported since the model is an extension of ANSI/SPARC, which provides such independence naturally. Location and replication transparencies are supported by the definition of the local and global conceptual schemas and the mapping in between. Network transparency, on the other hand, is supported by the definition of the global conceptual schema. The user

[4] In fact, in the first edition of this book which appeared in early 1990 and whose writing was completed in 1989, there wasn't a single mention of the term "client/server".

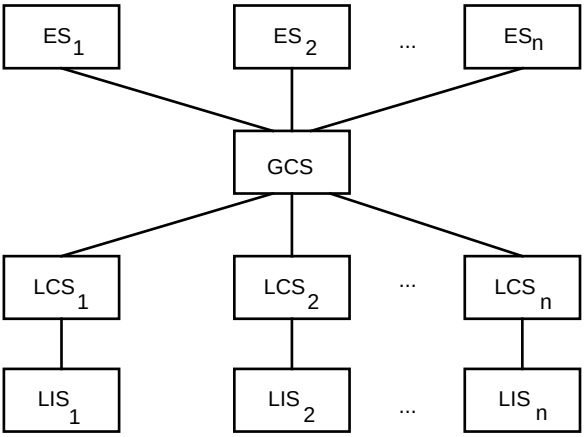

Fig. 1.14 Distributed Database Reference Architecture

queries data irrespective of its location or of which local component of the distributed database system will service it. As mentioned before, the distributed DBMS translates global queries into a group of local queries, which are executed by distributed DBMS components at different sites that communicate with one another.

The detailed components of a distributed DBMS are shown in Figure 1.15. One component handles the interaction with users, and another deals with the storage. The first major component, which we call the *user processor*, consists of four elements:

1. The *user interface handler* is responsible for interpreting user commands as they come in, and formatting the result data as it is sent to the user.
2. The *semantic data controller* uses the integrity constraints and authorizations that are defined as part of the global conceptual schema to check if the user query can be processed. This component, which is studied in detail in Chapter 5, is also responsible for authorization and other functions.
3. The *global query optimizer and decomposer* determines an execution strategy to minimize a cost function, and translates the global queries into local ones using the global and local conceptual schemas as well as the global directory. The global query optimizer is responsible, among other things, for generating the best strategy to execute distributed join operations. These issues are discussed in Chapters 6 through 8.
4. The *distributed execution monitor* coordinates the distributed execution of the user request. The execution monitor is also called the *distributed transaction manager*. In executing queries in a distributed fashion, the execution monitors at various sites may, and usually do, communicate with one another.

The second major component of a distributed DBMS is the *data processor* and consists of three elements:

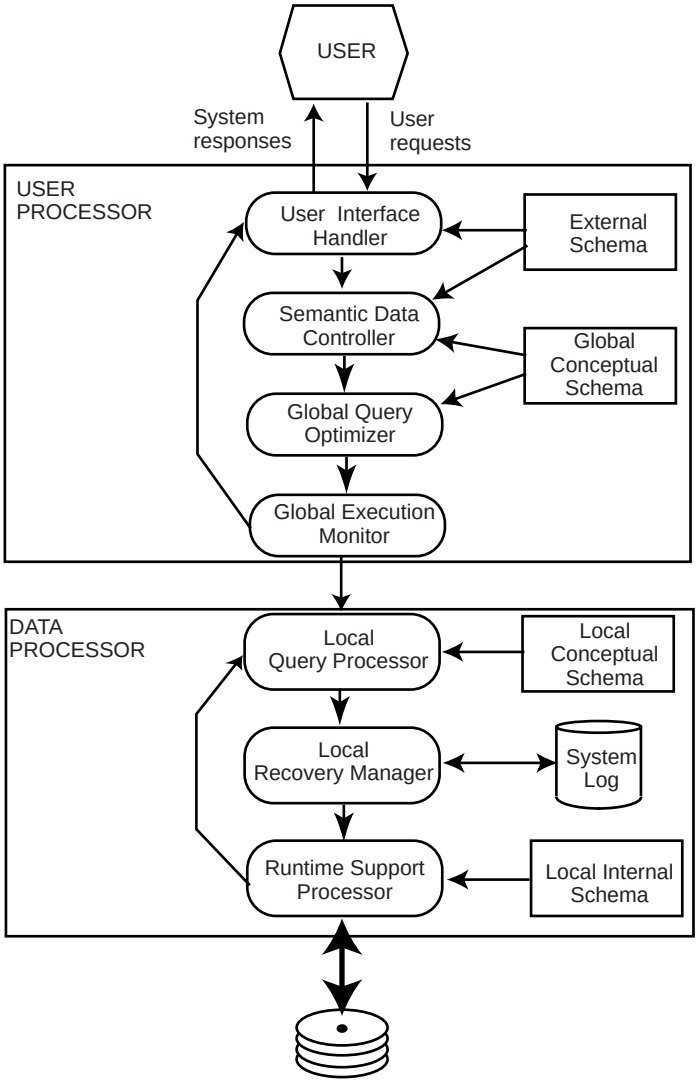

Fig. 1.15 Components of a Distributed DBMS

1. The *local query optimizer*, which actually acts as the *access path selector*, is responsible for choosing the best access path[5] to access any data item (touched upon briefly in Chapter 8).
2. The *local recovery manager* is responsible for making sure that the local database remains consistent even when failures occur (Chapter 12).
3. The *run-time support processor* physically accesses the database according to the physical commands in the schedule generated by the query optimizer. The run-time support processor is the interface to the operating system and contains the *database buffer* (or *cache*) *manager*, which is responsible for maintaining the main memory buffers and managing the data accesses.

It is important to note, at this point, that our use of the terms "user processor" and "data processor" does not imply a functional division similar to client/server systems. These divisions are merely organizational and there is no suggestion that they should be placed on different machines. In peer-to-peer systems, one expects to find both the user processor modules and the data processor modules on each machine. However, there have been suggestions to separate "query-only sites" in a system from full-functionality ones. In this case, the former sites would only need to have the user processor.

In client/server systems where there is a single server, the client has the user interface manager while the server has all of the data processor functionality as well as semantic data controller; there is no need for the global query optimizer or the global execution monitor. If there are multiple servers and the home server approach described in the previous section is employed, then each server hosts all of the modules except the user interface manager that resides on the client. If, however, each client is expected to contact individual servers on its own, then, most likely, the clients will host the full user processor functionality while the data processor functionality resides in the servers.

1.7.10 Multidatabase System Architecture

Multidatabase systems (MDBS) represent the case where individual DBMSs (whether distributed or not) are fully autonomous and have no concept of cooperation; they may not even "know" of each other's existence or how to talk to each other. Our focus is, naturally, on distributed MDBSs, which is what the term will refer to in the remainder. In most current literature, one finds the term *data integration system* used instead. We avoid using that term since data integration systems consider non-database data sources as well. Our focus is strictly on databases. We discuss these systems and their relationship to database integration in Chapter 4. We note, however, that there is considerable variability of the use of the term "multidatabase" in literature. In this

[5] The term *access path* refers to the data structures and the algorithms that are used to access the data. A typical access path, for example, is an index on one or more attributes of a relation.

book, we use it consistently as defined above, which may devitate from its use in some of the existing literature.

The differences in the level of autonomy between the distributed multi-DBMSs and distributed DBMSs are also reflected in their architectural models. The fundamental difference relates to the definition of the global conceptual schema. In the case of logically integrated distributed DBMSs, the global conceptual schema defines the conceptual view of the *entire* database, while in the case of distributed multi-DBMSs, it represents only the collection of *some* of the local databases that each local DBMS wants to share. The individual DBMSs may choose to make some of their data available for access by others (i.e., federated database architectures) by defining an *export schema* [Heimbigner and McLeod, 1985]. Thus the definition of a *global database* is different in MDBSs than in distributed DBMSs. In the latter, the global database is equal to the union of local databases, whereas in the former it is only a (possibly proper) subset of the same union. In a MDBS, the GCS (which is also called a mediated *schema*) is defined by integrating either the external schemas of local autonomous databases or (possibly parts of their) local conceptual schemas.

Furthermore, users of a local DBMS define their own views on the local database and do not need to change their applications if they do not want to access data from another database. This is again an issue of autonomy.

Designing the global conceptual schema in multidatabase systems involves the integration of either the local conceptual schemas or the local external schemas (Figure 1.16). A major difference between the design of the GCS in multi-DBMSs and in logically integrated distributed DBMSs is that in the former the mapping is from local conceptual schemas to a global schema. In the latter, however, mapping is in the reverse direction. As we discuss in Chapters 3 and 4, this is because the design in the former is usually a bottom-up process, whereas in the latter it is usually a top-down procedure. Furthermore, if heterogeneity exists in the multidatabase system, a canonical data model has to be found to define the GCS.

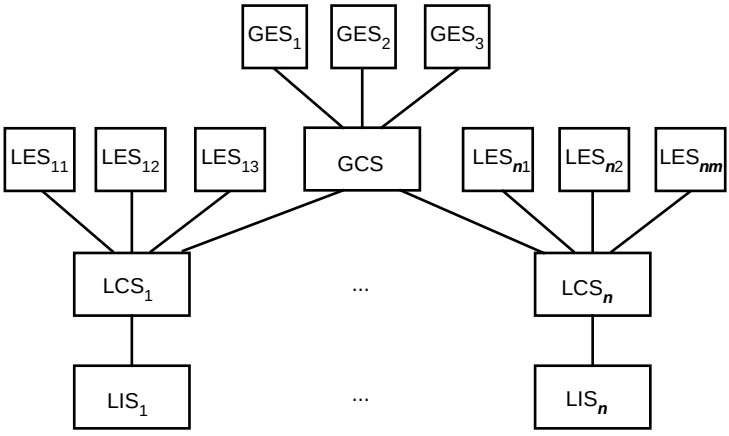

Fig. 1.16 MDBS Architecture with a GCS

Once the GCS has been designed, views over the global schema can be defined for users who require global access. It is not necessary for the GES and GCS to be defined using the same data model and language; whether they do or not determines whether the system is homogeneous or heterogeneous.

If heterogeneity exists in the system, then two implementation alternatives exist: unilingual and multilingual. A *unilingual* multi-DBMS requires the users to utilize possibly different data models and languages when both a local database and the global database are accessed. The identifying characteristic of unilingual systems is that any application that accesses data from multiple databases must do so by means of an external view that is defined on the global conceptual schema. This means that the user of the global database is effectively a different user than those who access only a local database, utilizing a different data model and a different data language.

An alternative is *multilingual* architecture, where the basic philosophy is to permit each user to access the global database (i.e., data from other databases) by means of an external schema, defined using the language of the user's local DBMS. The GCS definition is quite similar in the multilingual architecture and the unilingual approach, the major difference being the definition of the external schemas, which are described in the language of the external schemas of the local database. Assuming that the definition is purely local, a query issued according to a particular schema is handled exactly as any query in the centralized DBMSs. Queries against the global database are made using the language of the local DBMS, but they generally require some processing to be mapped to the global conceptual schema.

The component-based architectural model of a multi-DBMS is significantly different from a distributed DBMS. The fundamental difference is the existence of full-fledged DBMSs, each of which manages a different database. The MDBS provides a layer of software that runs on top of these individual DBMSs and provides users with the facilities of accessing various databases (Figure 1.17). Note that in a distributed MDBS, the multi-DBMS layer may run on multiple sites or there may be central site where those services are offered. Also note that as far as the individual DBMSs are concerned, the MDBS layer is simply another application that submits requests and receives answers.

A popular implementation architecture for MDBSs is the mediator/wrapper approach (Figure 1.18) [Wiederhold, 1992]. A *mediator* "is a software module that exploits encoded knowledge about certain sets or subsets of data to create information for a higher layer of applications." Thus, each mediator performs a particular function with clearly defined interfaces. Using this architecture to implement a MDBS, each module in the multi-DBMS layer of Figure 1.17 is realized as a mediator. Since mediators can be built on top of other mediators, it is possible to construct a layered implementation. In mapping this architecture to the datalogical view of Figure 1.16, the mediator level implements the GCS. It is this level that handles user queries over the GCS and performs the MDBS functionality.

The mediators typically operate using a common data model and interface language. To deal with potential heterogeneities of the source DBMSs, *wrappers* are implemented whose task is to provide a mapping between a source DBMSs view and the mediators' view. For example, if the source DBMS is a relational one, but the

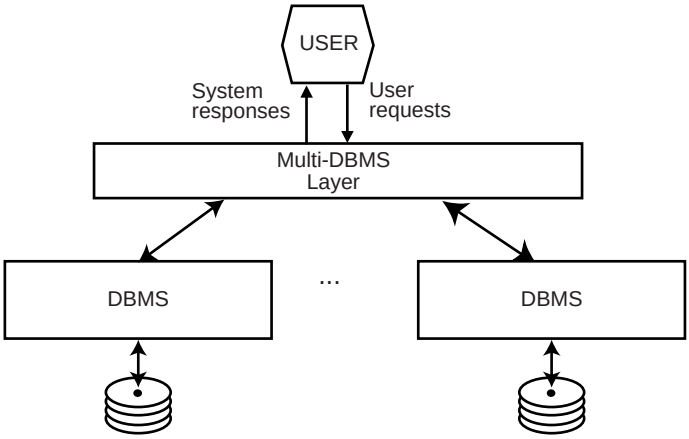

Fig. 1.17 Components of an MDBS

mediator implementations are object-oriented, the required mappings are established by the wrappers. The exact role and function of mediators differ from one implementation to another. In some cases, thin mediators have been implemented who do nothing more than translation. In other cases, wrappers take over the execution of some of the query functionality.

One can view the collection of mediators as a middleware layer that provides services above the source systems. Middleware is a topic that has been the subject of significant study in the past decade and very sophisticated middleware systems have been developed that provide advanced services for development of distributed applications. The mediators that we discuss only represent a subset of the functionality provided by these systems.

1.8 Bibliographic Notes

There are not many books on distributed DBMSs. Ceri and Pelagatti's book [Ceri and Pelagatti, 1983] was the first on this topic though it is now dated. The book by Bell and Grimson [Bell and Grimson, 1992] also provides an overview of the topics addressed here. In addition, almost every database book now has a chapter on distributed DBMSs. A brief overview of the technology is provided in [Özsu and Valduriez, 1997]. Our papers [Özsu and Valduriez, 1994, 1991] provide discussions of the state-of-the-art at the time they were written.

Database design is discussed in an introductory manner in [Levin and Morgan, 1975] and more comprehensively in [Ceri et al., 1987]. A survey of the file distribution algorithms is given in [Dowdy and Foster, 1982]. Directory management has not been considered in detail in the research community, but general techniques can be found in Chu and Nahouraii [1975] and [Chu, 1976]. A survey of query processing

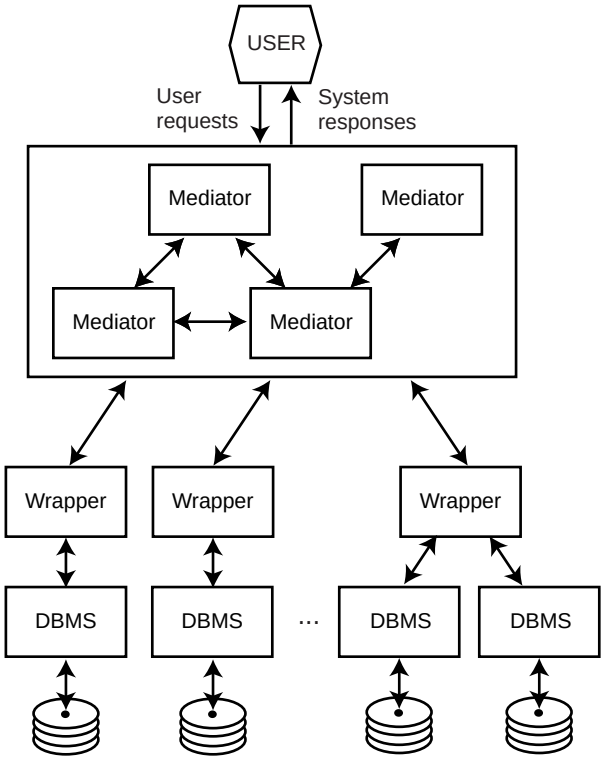

Fig. 1.18 Mediator/Wrapper Architecture

techniques can be found in [Sacco and Yao, 1982]. Concurrency control algorithms are reviewed in [Bernstein and Goodman, 1981] and [Bernstein et al., 1987]. Deadlock management has also been the subject of extensive research; an introductory paper is [Isloor and Marsland, 1980] and a widely quoted paper is [Obermarck, 1982]. For deadlock detection, good surveys are [Knapp, 1987] and [Elmagarmid, 1986]. Reliability is one of the issues discussed in [Gray, 1979], which is one of the landmark papers in the field. Other important papers on this topic are [Verhofstadt, 1978] and [Härder and Reuter, 1983]. [Gray, 1979] is also the first paper discussing the issues of operating system support for distributed databases; the same topic is addressed in [Stonebraker, 1981]. Unfortunately, both papers emphasize centralized database systems.

There have been a number of architectural framework proposals. Some of the interesting ones include Schreiber's quite detailed extension of the ANSI/SPARC framework which attempts to accommodate heterogeneity of the data models [Schreiber, 1977], and the proposal by Mohan and Yeh [Mohan and Yeh, 1978]. As expected, these date back to the early days of the introduction of distributed DBMS technology. The detailed component-wise system architecture given in Figure 1.15 derives from

[Rahimi, 1987]. An alternative to the classification that we provide in Figure 1.10 can be found in [Sheth and Larson, 1990].

Most of the discussion on architectural models for multi-DBMSs is from [Özsu and Barker, 1990]. Other architectural discussions on multi-DBMSs are given in [Gligor and Luckenbaugh, 1984], [Litwin, 1988], and [Sheth and Larson, 1990]. All of these papers provide overview discussions of various prototype and commercial systems. An excellent overview of heterogeneous and federated database systems is [Sheth and Larson, 1990].

Chapter 2
Background

As indicated in the previous chapter, there are two technological bases for distributed database technology: database management and computer networks. In this chapter, we provide an overview of the concepts in these two fields that are more important from the perspective of distributed database technology.

2.1 Overview of Relational DBMS

The aim of this section is to define the terminology and framework used in subsequent chapters, since most of the distributed database technology has been developed using the relational model. In later chapters, when appropriate, we introduce other models. Our focus here is on the language and operators.

2.1.1 Relational Database Concepts

A *database* is a structured collection of data related to some real-life phenomena that we are trying to model. A *relational database* is one where the database structure is in the form of tables. Formally, a relation R defined over n sets $D_1, D_2, \ldots, D_n$ (not necessarily distinct) is a set of *n-tuples* (or simply *tuples*) $\langle d_1, d_2, \ldots, d_n \rangle$ such that $d_1 \in D_1, d_2 \in D_2, \ldots, d_n \in D_n$.

Example 2.1. As an example we use a database that models an engineering company. The entities to be modeled are the *employees* (EMP) and *project*s (PROJ). For each employee, we would like to keep track of the employee number (ENO), name (ENAME), title in the company (TITLE), salary (SAL), identification number of the project(s) the employee is working on (PNO), responsibility within the project (RESP), and duration of the assignment to the project (DUR) in months. Similarly, for each project we would like to store the project number (PNO), the project name (PNAME), and the project budget (BUDGET).

M.T. Özsu and P. Valduriez, *Principles of Distributed Database Systems: Third Edition*,
DOI 10.1007/978-1-4419-8834-8_2,

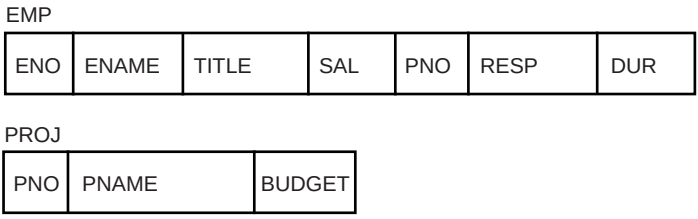

Fig. 2.1 Sample Database Scheme

The *relation schemas* for this database can be defined as follows:

EMP(ENO, ENAME, TITLE, SAL, PNO, RESP, DUR)

PROJ(PNO,PNAME, BUDGET)

In relation scheme EMP, there are seven *attributes*: ENO, ENAME, TITLE, SAL, PNO, RESP, DUR. The values of ENO come from the *domain* of all valid employee numbers, say D_1, the values of ENAME come from the domain of all valid names, say D_2, and so on. Note that each attribute of each relation does not have to come from a distinct domain. Various attributes within a relation or from a number of relations may be defined over the same domain. ♦

The *key* of a relation scheme is the minimum non-empty subset of its attributes such that the values of the attributes comprising the key uniquely identify each tuple of the relation. The attributes that make up key are called *prime* attributes. The superset of a key is usually called a *superkey*. Thus in our example the key of PROJ is PNO, and that of EMP is the set (ENO, PNO). Each relation has at least one key. Sometimes, there may be more than one possibility for the key. In such cases, each alternative is considered a *candidate key*, and one of the candidate keys is chosen as the *primary key*, which we denote by underlining. The number of attributes of a relation defines its *degree*, whereas the number of tuples of the relation defines its *cardinality*.

In tabular form, the example database consists of two tables, as shown in Figure 2.1. The columns of the tables correspond to the attributes of the relations; if there were any information entered as the rows, they would correspond to the tuples. The empty table, showing the structure of the table, corresponds to the *relation schema*; when the table is filled with rows, it corresponds to a *relation instance*. Since the information within a table varies over time, many instances can be generated from one relation scheme. Note that from now on, the term *relation* refers to a relation instance. In Figure 2.2 we depict instances of the two relations that are defined in Figure 2.1.

An attribute value may be undefined. This lack of definition may have various interpretations, the most common being “unknown” or “not applicable”. This special value of the attribute is generally referred to as the *null value*. The representation of a null value must be different from any other domain value, and special care should be given to differentiate it from zero. For example, value “0” for attribute DUR is

EMP

ENO	ENAME	TITLE	SAL	PNO	RESP	DUR
E1	J. Doe	Elect. Eng.	40000	P1	Manager	12
E2	M. Smith	Analyst	34000	P1	Analyst	24
E2	M. Smith	Analyst	34000	P2	Analyst	6
E3	A. Lee	Mech. Eng.	27000	P3	Consultant	10
E3	A. Lee	Mech. Eng.	27000	P4	Engineer	48
E4	J. Miller	Programmer	24000	P2	Programmer	18
E5	B. Casey	Syst. Anal.	34000	P2	Manager	24
E6	L. Chu	Elect. Eng.	40000	P4	Manager	48
E7	R. Davis	Mech. Eng.	27000	P3	Engineer	36
E8	J. Jones	Syst. Anal.	34000	P3	Manager	40

PROJ

PNO	PNAME	BUDGET
P1	Instrumentation	150000
P2	Database Develop.	135000
P3	CAD/CAM	250000
P4	Maintenance	310000

Fig. 2.2 Sample Database Instance

known information (e.g., in the case of a newly hired employee), while value "null" for DUR means unknown. Supporting null values is an important feature necessary to deal with *maybe* queries [Codd, 1979].

2.1.2 Normalization

The aim of normalization is to eliminate various anomalies (or undesirable aspects) of a relation in order to obtain "better" relations. The following four problems might exist in a relation scheme:

1. *Repetition anomaly*. Certain information may be repeated unnecessarily. Consider, for example, the EMP relation in Figure 2.2. The name, title, and salary of an employee are repeated for each project on which this person serves. This is obviously a waste of storage and is contrary to the spirit of databases.

2. *Update anomaly*. As a consequence of the repetition of data, performing updates may be troublesome. For example, if the salary of an employee changes, multiple tuples have to be updated to reflect this change.
3. *Insertion anomaly*. It may not be possible to add new information to the database. For example, when a new employee joins the company, we cannot add personal information (name, title, salary) to the EMP relation unless an appointment to a project is made. This is because the key of EMP includes the attribute PNO, and null values cannot be part of the key.
4. *Deletion anomaly*. This is the converse of the insertion anomaly. If an employee works on only one project, and that project is terminated, it is not possible to delete the project information from the EMP relation. To do so would result in deleting the only tuple about the employee, thereby resulting in the loss of personal information we might want to retain.

Normalization transforms arbitrary relation schemes into ones without these problems. A relation with one or more of the above mentioned anomalies is split into two or more relations of a higher *normal form*. A relation is said to be in a normal form if it satisfies the conditions associated with that normal form. Codd initially defined the *first*, *second*, and *third* normal forms (1NF, 2NF, and 3NF, respectively). Boyce and Codd [Codd, 1974] later defined a modified version of the third normal form, commonly known as the *Boyce-Codd normal form* (BCNF). This was followed by the definition of the *fourth* (4NF) [Fagin, 1977] and *fifth* normal forms (5NF) [Fagin, 1979].

The normal forms are based on certain dependency structures. BCNF and lower normal forms are based on *functional dependencies* (FDs), 4NF is based on *multivalued dependencies*, and 5NF is based on *projection-join dependencies*. We only introduce functional dependency, since that is the only relevant one for the example we are considering.

Let R be a relation defined over the set of attributes $A = \{A_1, A_2, \ldots, A_n\}$ and let $X \subset A$, $Y \subset A$. If for each value of X in R, there is only one associated Y value, we say that "X *functionally determines* Y" or that "Y is *functionally dependent* on X." Notationally, this is shown as $X \rightarrow Y$. The key of a relation functionally determines the non-key attributes of the same relation.

Example 2.2. For example, in the PROJ relation of Example 2.1 (one can observe these in Figure 2.2 as well), the valid FD is

$$\text{PNO} \rightarrow (\text{PNAME, BUDGET})$$

In the EMP relation we have

$$(\text{ENO, PNO}) \rightarrow (\text{ENAME,TITLE,SAL,RESP,DUR})$$

This last FD is not the only FD in EMP, however. If each employee is given unique employee numbers, we can write

ENO → (ENAME, TITLE, SAL)

(ENO, PNO) → (RESP, DUR)

It may also happen that the salary for a given position is fixed, which gives rise to the FD

TITLE → SAL

♦

We do not discuss the normal forms or the normalization algorithms in detail; these can be found in database textbooks. The following example shows the result of normalization on the sample database that we introduced in Example 2.1.

Example 2.3. The following set of relation schemes are normalized into BCNF with respect to the functional dependencies defined over the relations.

EMP(ENO, ENAME, TITLE)

PAY(TITLE, SAL)

PROJ(PNO, PNAME, BUDGET)

ASG(ENO, PNO, RESP, DUR)

The normalized instances of these relations are shown in Figure 2.3. ♦

2.1.3 Relational Data Languages

Data manipulation languages developed for the relational model (commonly called *query languages*) fall into two fundamental groups: *relational algebra* languages and *relational calculus* languages. The difference between them is based on how the user query is formulated. The relational algebra is procedural in that the user is expected to specify, using certain high-level operators, how the result is to be obtained. The relational calculus, on the other hand, is non-procedural; the user only specifies the relationships that should hold in the result. Both of these languages were originally proposed by Codd [1970], who also proved that they were equivalent in terms of expressive power [Codd, 1972].

2.1.3.1 Relational Algebra

Relational algebra consists of a set of operators that operate on relations. Each operator takes one or two relations as operands and produces a result relation, which, in turn, may be an operand to another operator. These operations permit the querying and updating of a relational database.

EMP

ENO	ENAME	TITLE
E1	J. Doe	Elect. Eng
E2	M. Smith	Syst. Anal.
E3	A. Lee	Mech. Eng.
E4	J. Miller	Programmer
E5	B. Casey	Syst. Anal.
E6	L. Chu	Elect. Eng.
E7	R. Davis	Mech. Eng.
E8	J. Jones	Syst. Anal.

ASG

ENO	PNO	RESP	DUR
E1	P1	Manager	12
E2	P1	Analyst	24
E2	P2	Analyst	6
E3	P3	Consultant	10
E3	P4	Engineer	48
E4	P2	Programmer	18
E5	P2	Manager	24
E6	P4	Manager	48
E7	P3	Engineer	36
E8	P3	Manager	40

PROJ

PNO	PNAME	BUDGET
P1	Instrumentation	150000
P2	Database Develop.	135000
P3	CAD/CAM	250000
P4	Maintenance	310000

PAY

TITLE	SAL
Elect. Eng.	40000
Syst. Anal.	34000
Mech. Eng.	27000
Programmer	24000

Fig. 2.3 Normalized Relations

There are five fundamental relational algebra operators and five others that can be defined in terms of these. The fundamental operators are *selection*, *projection*, *union*, *set difference*, and *Cartesian product*. The first two of these operators are unary operators, and the last three are binary operators. The additional operators that can be defined in terms of these fundamental operators are *intersection*, $\theta - join$, *natural join*, *semijoin* and *division*. In practice, relational algebra is extended with operators for grouping or sorting the results, and for performing arithmetic and aggregate functions. Other operators, such as *outer join* and *transitive closure*, are sometimes used as well to provide additional functionality. We only discuss the more common operators.

The operands of some of the binary relations should be *union compatible*. Two relations R and S are union compatible if and only if they are of the same degree and the i-th attribute of each is defined over the same domain. The second part of the definition holds, obviously, only when the attributes of a relation are identified by their relative positions within the relation and not by their names. If relative ordering of attributes is not important, it is necessary to replace the second part of the definition by the phrase "the corresponding attributes of the two relations should be defined over the same domain." The correspondence is defined rather loosely here.

Many operator definitions refer to "formula", which also appears in relational calculus expressions we discuss later. Thus, let us define precisely, at this point, what we mean by a formula. We define a formula within the context of first-order predicate

calculus (since we use that formalism later), and follow the notation of Gallaire et al. [1984]. First-order predicate calculus is based on a *symbol alphabet* that consists of (1) variables, constants, functions, and predicate symbols; (2) parentheses; (3) the logical connectors $\wedge$ (and), $\vee$ (or), $\neg$ (not), $\rightarrow$ (implication), and $\leftrightarrow$ (equivalence); and (4) quantifiers $\forall$ (for all) and $\exists$ (there exists). A *term* is either a constant or a variable. Recursively, if f is an n-ary function and $t_1, \ldots, t_n$ are terms, $f(t_1, \ldots, t_n)$ is also a term. An *atomic formula* is of the form $P(t_1, \ldots, t_n)$, where P is an n-ary predicate symbol and the t_i's are terms. A *well-formed formula* (*wff*) can be defined recursively as follows: If w_i and w_j are wffs, then (w_i), $\neg(w_i), (w_i) \wedge (w_j), (w_i) \vee (w_j), (w_i) \rightarrow (w_j)$, and $(w_i) \leftrightarrow (w_j)$ are all wffs. Variables in a wff may be *free* or they may be *bound* by one of the two quantifiers.

Selection.

Selection produces a horizontal subset of a given relation. The subset consists of all the tuples that satisfy a formula (condition). The selection from a relation R is

$$\sigma_F(R)$$

where R is the relation and F is a formula.

The formula in the selection operation is called a *selection predicate* and is an atomic formula whose terms are of the form $A\theta c$, where A is an attribute of R and θ is one of the arithmetic comparison operators $<, >, =, \neq, \leq,$ and $\geq$. The terms can be connected by the logical connectors $\wedge, \vee$, and $\neg$. Furthermore, the selection predicate does not contain any quantifiers.

Example 2.4. Consider the relation EMP shown in Figure 2.3. The result of selecting those tuples for electrical engineers is shown in Figure 2.4. ♦

$\sigma_{\text{TITLE="Elect. Eng."}}(\text{EMP})$

ENO	ENAME	TITLE
E1	J. Doe	Elect. Eng
E6	L. Chu	Elect. Eng.

Fig. 2.4 Result of Selection

Projection.

Projection produces a vertical subset of a relation. The result relation contains only those attributes of the original relation over which projection is performed. Thus the degree of the result is less than or equal to the degree of the original relation.

The projection of relation R over attributes A and B is denoted as

$$\Pi_{A,B}(R)$$

Note that the result of a projection might contain tuples that are identical. In that case the duplicate tuples may be deleted from the result relation. It is possible to specify projection with or without duplicate elimination.

Example 2.5. The projection of relation PROJ shown in Figure 2.3 over attributes PNO and BUDGET is depicted in Figure 2.5. ♦

$\Pi_{\text{PNO,BUDGET}}$ (PROJ)

PNO	BUDGET
P1	150000
P2	135000
P3	250000
P4	310000

Fig. 2.5 Result of Projection

Union.

The union of two relations R and S (denoted as $R \cup S$) is the set of all tuples that are in R, or in S, or in both. We should note that R and S should be union compatible. As in the case of projection, the duplicate tuples are normally eliminated. Union may be used to insert new tuples into an existing relation, where these tuples form one of the operand relations.

Set Difference.

The set difference of two relations R and S ($R - S$) is the set of all tuples that are in R but not in S. In this case, not only should R and S be union compatible, but the operation is also asymmetric (i.e., $R - S \neq S - R$). This operation allows the

EMP x PAY

ENO	ENAME	EMP.TITLE	PAY.TITLE	SAL
E1	J. Doe	Elect. Eng.	Elect. Eng.	40000
E1	J. Doe	Elect. Eng.	Syst. Anal.	34000
E1	J. Doe	Elect. Eng.	Mech. Eng.	27000
E1	J. Doe	Elect. Eng.	Programmer	24000
E2	M. Smith	Syst. Anal.	Elect. Eng.	40000
E2	M. Smith	Syst. Anal.	Syst. Anal.	34000
E2	M. Smith	Syst. Anal.	Mech. Eng.	27000
E2	M. Smith	Syst. Anal.	Programmer	24000
E3	A. Lee	Mech. Eng.	Elect. Eng.	40000
E3	A. Lee	Mech. Eng.	Syst. Anal.	34000
E3	A. Lee	Mech. Eng.	Mech. Eng.	27000
E3	A. Lee	Mech. Eng.	Programmer	24000
≈	≈	≈	≈	≈
E8	J. Jones	Syst. Anal.	Elect. Eng.	40000
E8	J. Jones	Syst. Anal.	Syst. Anal.	34000
E8	J. Jones	Syst. Anal.	Mech. Eng.	27000
E8	J. Jones	Syst. Anal.	Programmer	24000

Fig. 2.6 Partial Result of Cartesian Product

deletion of tuples from a relation. Together with the union operation, we can perform modification of tuples by deletion followed by insertion.

Cartesian Product.

The Cartesian product of two relations R of degree k_1 and S of degree k_2 is the set of $(k_1 + k_2)$-tuples, where each result tuple is a concatenation of one tuple of R with one tuple of S, for all tuples of R and S. The Cartesian product of R and S is denoted as $R \times S$.

It is possible that the two relations might have attributes with the same name. In this case the attribute names are prefixed with the relation name so as to maintain the uniqueness of the attribute names within a relation.

Example 2.6. Consider relations EMP and PAY in Figure 2.3. EMP $\times$ PAY is shown in Figure 2.6. Note that the attribute TITLE, which is common to both relations, appears twice, prefixed with the relation name. ♦

Intersection.

Intersection of two relations R and S ($R \cap S$) consists of the set of all tuples that are in both R and S. In terms of the basic operators, it can be specified as follows:

$$R \cap S = R - (R - S)$$

θ-Join.

Join is a derivative of Cartesian product. There are various forms of join; the primary classification is between *inner join* and *outer join*. We first discuss inner join and its variants and then describe outer join.

The most general type of inner join is the θ-join. The θ-join of two relations R and S is denoted as

$$R \bowtie_F S$$

where F is a formula specifying the *join predicate*. A join predicate is specified similar to a selection predicate, except that the terms are of the form $R.A\theta S.B$, where A and B are attributes of R and S, respectively.

The join of two relations is equivalent to performing a selection, using the join predicate as the selection formula, over the Cartesian product of the two operand relations. Thus

$$R \bowtie_F S = \sigma_F(R \times S)$$

In the equivalence above, we should note that if F involves attributes of the two relations that are common to both of them, a projection is necessary to make sure that those attributes do not appear twice in the result.

Example 2.7. Let us consider that the EMP relation in Figure 2.3 and add two more tuples as depicted in Figure 2.7(a). Then Figure 2.7(b) shows the θ-join of relations EMP and ASG over the join predicate EMP.ENO=ASG.ENO.

The same result could have been obtained as

$$\text{EMP} \bowtie_{\text{EMP.ENO=ASG.ENO}} \text{ASG} = \Pi_{\text{ENO, ENAME, TITLE, SAL}}(\sigma_{\text{EMP.ENO =PAY.ENO}}(\text{EMP} \times \text{ASG}))$$

Notice that the result does not have tuples E9 and E10 since these employees have not yet been assigned to a project. Furthermore, the information about some employees (e.g., E2 and E3) who have been assigned to multiple projects appear more than once in the result. ♦

This example demonstrates a special case of θ-join which is called the *equi-join*. This is a case where the formula F only contains equality (=) as the arithmetic operator. It should be noted, however, that an equi-join does not have to be specified over a common attribute as the example above might suggest.

EMP

ENO	ENAME	TITLE
E1	J. Doe	Elect. Eng
E2	M. Smith	Syst. Anal.
E3	A. Lee	Mech. Eng.
E4	J. Miller	Programmer
E5	B. Casey	Syst. Anal.
E6	L. Chu	Elect. Eng.
E7	R. Davis	Mech. Eng.
E8	J. Jones	Syst. Anal.
E9	A. Hsu	Programmer
E10	T. Wong	Syst. Anal.

(a)

EMP $\bowtie_{\text{EMP.ENO=ASG.ENO}}$ ASG

ENO	ENAME	TITLE	PNO	RESP	DUR
E1	J. Doe	Elect. Eng.	P1	Manager	12
E2	M. Smith	Syst. Anal.	P1	Analyst	12
E2	M. Smith	Syst. Anal.	P2	Analyst	12
E3	A. Lee	Mech. Eng.	P3	Consultant	12
E3	A. Lee	Mech. Eng.	P4	Engineer	12
E4	J. Miller	Programmer	P2	Programmer	12
E5	J. Miller	Syst. Anal.	P2	Manager	12
E6	L. Chu	Elect. Eng.	P4	Manager	12
E7	R. Davis	Mech. Eng.	P3	Engineer	12
E8	J. Jones	Syst. Anal.	P3	Manager	12

(b)

Fig. 2.7 The Result of Join

A natural join is an equi-join of two relations over a specified attribute, more specifically, over attributes with the same domain. There is a difference, however, in that usually the attributes over which the natural join is performed appear only once in the result. A natural join is denoted as the join without the formula

$$R \bowtie_A S$$

where A is the attribute common to both R and S. We should note here that the natural join attribute may have different names in the two relations; what is required is that they come from the same domain. In this case the join is denoted as

$$R_A \bowtie_B S$$

where B is the corresponding join attribute of S.

Example 2.8. The join of EMP and ASG in Example 2.7 is actually a natural join. Here is another example – Figure 2.8 shows the natural join of relations EMP and PAY in Figure 2.3 over the attribute TITLE.

♦

Inner join requires the joined tuples from the two operand relations to satisfy the join predicate. In contrast, outer join does not have this requirement – tuples exist in the result relation regardless. Outer join can be of three types: left outer join ( ), right outer join ( ) and full outer join ( ). In the left outer join, the tuples from the left operand relation are always in the result, in the case of right outer join, the tuples from the right operand are always in the result, and in the case of full outer relation, tuples from both relations are always in the result. Outer join is useful in those cases where we wish to include information from one or both relations even if the do not satisfy the join predicate.

EMP ⋈ TITLE PAY

ENO	ENAME	TITLE	SAL
E1	J. Doe	Elect. Eng.	40000
E2	M. Smith	Analyst	34000
E3	A. Lee	Mech. Eng.	27000
E4	J. Miller	Programmer	24000
E5	B. Casey	Syst. Anal.	34000
E6	L. Chu	Elect. Eng.	40000
E7	R. Davis	Mech. Eng.	27000
E8	J. Jones	Syst. Anal.	34000

Fig. 2.8 The Result of Natural Join

Example 2.9. Consider the left outer join of EMP (as revised in Example 2.7) and ASG over attribute ENO (i.e., EMP ENO ASG). The result is given in Figure 2.9. Notice that the information about two employees, E9 and E10 are included in the result even thought they have not yet been assigned to a project with "Null" values for the attributes from the ASG relation. ♦

EMP ⟕ ENO ASG

ENO	ENAME	TITLE	PNO	RESP	DUR
E1	J. Doe	Elect. Eng.	P1	Manager	12
E2	M. Smith	Syst. Anal.	P1	Analyst	12
E2	M. Smith	Syst. Anal.	P2	Analyst	12
E3	A. Lee	Mech. Eng.	P3	Consultant	12
E3	A. Lee	Mech. Eng.	P4	Engineer	12
E4	J. Miller	Programmer	P2	Programmer	12
E5	J. Miller	Syst. Anal.	P2	Manager	12
E6	L. Chu	Elect. Eng.	P4	Manager	12
E7	R. Davis	Mech. Eng.	P3	Engineer	12
E8	J. Jones	Syst. Anal.	P3	Manager	12
E9	A. Hsu	Programmer	Null	Null	Null
E10	T. Wong	Syst. Anal.	Null	Null	Null

Fig. 2.9 The Result of Left Outer Join

Semijoin.

The semijoin of relation R, defined over the set of attributes A, by relation S, defined over the set of attributes B, is the subset of the tuples of R that participate in the join of R with S. It is denoted as $R \ltimes_F S$ (where F is a predicate as defined before) and can be obtained as follows:

$$\begin{aligned} R \ltimes_F S &= \Pi_A(R \bowtie_F S) = \Pi_A(R) \bowtie_F \Pi_{A \cap B}(S) \\ &= R \bowtie_F \Pi_{A \cap B}(S) \end{aligned}$$

The advantage of semijoin is that it decreases the number of tuples that need to be handled to form the join. In centralized database systems, this is important because it usually results in a decreased number of secondary storage accesses by making better use of the memory. It is even more important in distributed databases since it usually reduces the amount of data that needs to be transmitted between sites in order to evaluate a query. We talk about this in more detail in Chapters 3 and 8. At this point note that the operation is asymmetric (i.e., $R \ltimes_F S \neq S \ltimes_F R$).

Example 2.10. To demonstrate the difference between join and semijoin, let us consider the semijoin of EMP with PAY over the predicate EMP.TITLE = PAY.TITLE, that is,

$$\text{EMP} \ltimes_{\text{EMP.TITLE} = \text{PAY.TITLE}} \text{PAY}$$

The result of the operation is shown in Figure 2.10. We encourage readers to compare Figures 2.7 and 2.10 to see the difference between the join and the semijoin operations. Note that the resultant relation does not have the PAY attribute and is therefore smaller. ♦

EMP $\ltimes_{\text{EMP.TITLE=PAY.TITLE}}$ PAY

ENO	ENAME	TITLE
E1	J. Doe	Elect. Eng.
E2	M. Smith	Analyst
E3	A. Lee	Mech. Eng.
E4	J. Miller	Programmer
E5	B. Casey	Syst. Anal.
E6	L. Chu	Elect. Eng.
E7	R. Davis	Mech. Eng.
E8	J. Jones	Syst. Anal.

Fig. 2.10 The Result of Semijoin

Division.

The division of relation R of degree r with relation S of degree s (where $r > s$ and $s \neq 0$) is the set of $(r-s)$-tuples t such that for all s-tuples u in S, the tuple tu is in R. The division operation is denoted as $R \div S$ and can be specified in terms of the fundamental operators as follows:

$$R \div S = \Pi_{\bar{A}}(R) - \Pi_{\bar{A}}((\Pi_{\bar{A}}(R) \times S) - R)$$

where $\bar{A}$ is the set of attributes of R that are not in S [i.e., the $(r-s)$-tuples].

Example 2.11. Assume that we have a modified version of the ASG relation (call it ASG′) depicted in Figure 2.11a and defined as follows:

$$\text{ASG}' = \Pi_{\text{ENO,PNO}}\ (\text{ASG}) \bowtie_{\text{PNO}} \text{PROJ}$$

If one wants to find the employee numbers of those employees who are assigned to all the projects that have a budget greater than \$200,000, it is necessary to divide ASG′ with a restricted version of PROJ, called PROJ′ (see Figure 2.11b). The result of division (ASG′ ÷ PROJ′) is shown in Figure 2.11c.

The keyword in the query above is "*all*." This rules out the possibility of doing a selection on ASG′ to find the necessary tuples, since that would only give those which correspond to employees working on *some* project with a budget greater than \$200,000, not those who work on all projects. Note that the result contains only the tuple ⟨E3⟩ since the tuples ⟨E3, P3, CAD/CAM, 250000⟩ and ⟨E3, P4, Maintenance, 310000⟩ both exist in ASG′. On the other hand, for example, ⟨E7⟩ is not in the result, since even though the tuple ⟨E7, P3, CAD/CAM, 250000⟩ is in ASG′, the tuple ⟨E7, P4, Maintenance, 310000⟩ is not. ♦

Since all operations take relations as input and produce relations as outputs, we can nest operations using a parenthesized notation and represent relational algebra programs. The parentheses indicate the order of execution. The following are a few examples that demonstrate the issue.

Example 2.12. Consider the relations of Figure 2.3. The retrieval query

"Find the names of employees working on the CAD/CAM project"

can be answered by the relational algebra program

$$\Pi_{\text{ENAME}}(((\sigma_{\text{PNAME = "CAD/CAM"}}\ \text{PROJ}) \bowtie_{\text{PNO}} \text{ASG}) \bowtie_{\text{ENO}} \text{EMP})$$

The order of execution is: the selection on PROJ, followed by the join with ASG, followed by the join with EMP, and finally the project on ENAME.

An equivalent program where the size of the intermediate relations is smaller is

$$\Pi_{\text{ENAME}}\ (\text{EMP} \ltimes_{\text{ENO}} (\Pi_{\text{ENO}} (\text{ASG} \ltimes_{\text{PNO}} (\sigma_{\text{PNAME= "CAD/CAM"}}\ \text{PROJ}))))$$

♦

ASG'

ENO	PNO	PNAME	BUDGET
E1	P1	Instrumentation	150000
E2	P1	Instrumentation	150000
E2	P2	Database Develop.	135000
E3	P3	CAD/CAM	250000
E3	P4	Maintenance	310000
E4	P2	Database Develop.	135000
E5	P2	Database Develop.	135000
E6	P4	Maintenance	310000
E7	P3	CAD/CAM	250000
E8	P3	CAD/CAM	250000

(a)

PROJ'

PNO	PNAME	BUDGET
P3	CAD/CAM	250000
P4	Maintenance	310000

(b)

(ASG' ÷ PROJ')

ENO
E3

(c)

Fig. 2.11 The Result of Division

Example 2.13. The update query

"Replace the salary of programmers by $25,000"

can be computed by

$$(\text{PAY} - (\sigma_{\text{TITLE} = \text{“Programmer”}} \text{PAY})) \cup (\langle \text{Programmer}, 25000 \rangle)$$

♦

2.1.3.2 Relational Calculus

In relational calculus-based languages, instead of specifying *how* to obtain the result, one specifies *what* the result is by stating the relationship that is supposed to hold for the result. Relational calculus languages fall into two groups: *tuple relational calculus* and *domain relational calculus*. The difference between the two is in terms

of the primitive variable used in specifying the queries. We briefly review these two types of languages.

Relational calculus languages have a solid theoretical foundation since they are based on first-order predicate logic as we discussed before. Semantics is given to formulas by interpreting them as assertions on the database. A relational database can be viewed as a collection of tuples or a collection of domains. Tuple relational calculus interprets a variable in a formula as a tuple of a relation, whereas domain relational calculus interprets a variable as the value of a domain.

Tuple relational calculus.

The primitive variable used in tuple relational calculus is a *tuple variable* which specifies a tuple of a relation. In other words, it ranges over the tuples of a relation. Tuple calculus is the original relational calculus developed by Codd [1970].

In tuple relational calculus queries are specified as $\{t|F(t)\}$, where t is a tuple variable and F is a well-formed formula. The atomic formulas are of two forms:

1. *Tuple-variable membership expressions*. If t is a tuple variable ranging over the tuples of relation R (predicate symbol), the expression "tuple t belongs to relation R" is an atomic formula, which is usually specified as $R.t$ or $R(t)$.
2. *Conditions*. These can be defined as follows:

   (a) $s[A]\theta t[B]$, where s and t are tuple variables and A and B are components of s and t, respectively. θ is one of the arithmetic comparison operators $<$, $>$, $=$, $\neq$, $\leq$, and $\geq$. This condition specifies that component A of s stands in relation θ to the B component of t: for example, $s[\text{SAL}] > t[\text{SAL}]$.

   (b) $s[A]\theta c$, where s, A, and θ are as defined above and c is a constant. For example, $s[\text{ENAME}] = \text{"Smith"}$.

Note that A is defined as a component of the tuple variable s. Since the range of s is a relation instance, say S, it is obvious that component A of s corresponds to attribute A of relation S. The same thing is obviously true for B.

There are many languages that are based on relational tuple calculus, the most popular ones being SQL[1] [Date, 1987] and QUEL [Stonebraker et al., 1976]. SQL is now an international standard (actually, the only one) with various versions released: SQL1 was released in 1986, modifications to SQL1 were included in the 1989 version, SQL2 was issued in 1992, and SQL3, with object-oriented language extensions, was released in 1999.

[1] Sometimes SQL is cited as lying somewhere between relational algebra and relational calculus. Its originators called it a "mapping language." However, it follows the tuple calculus definition quite closely; hence we classify it as such.

SQL provides a uniform approach to data manipulation (retrieval, update), data definition (schema manipulation), and control (authorization, integrity, etc.). We limit ourselves to the expression, in SQL, of the queries in Examples 2.14 and 2.15.

Example 2.14. The query from Example 2.12,

"Find the names of employees working on the CAD/CAM project"

can be expressed as follows:

```
SELECT  EMP.ENAME
FROM    EMP,ASG,PROJ
WHERE   EMP.ENO = ASG.ENO
AND     ASG.PNO = PROJ.PNO
AND     PROJ.PNAME = "CAD/CAM"
```

♦

Note that a retrieval query generates a new relation similar to the relational algebra operations.

Example 2.15. The update query of Example 2.13,

"Replace the salary of programmers by $25,000"

is expressed as

```
UPDATE  PAY
SET     SAL = 25000
WHERE   PAY.TITLE = "Programmer"
```

♦

Domain relational calculus.

The domain relational calculus was first proposed by Lacroix and Pirotte [1977]. The fundamental difference between a tuple relational language and a domain relational language is the use of a *domain variable* in the latter. A domain variable ranges over the values in a domain and specifies a component of a tuple. In other words, the range of a domain variable consists of the domains over which the relation is defined. The wffs are formulated accordingly. The queries are specified in the following form:

$$x_1, x_2, \ldots, x_n | F(x_1, x_2, \ldots, x_n)$$

where F is a wff in which $x_1, \ldots, x_n$ are the free variables.

The success of domain relational calculus languages is due mainly to QBE [Zloof, 1977], which is a visual application of domain calculus. QBE, designed only for interactive use from a visual terminal, is user friendly. The basic concept is an *example*: the user formulates queries by providing a possible example of the answer. Typing relation names triggers the printing, on screen, of their schemes. Then, by supplying keywords into the columns (domains), the user specifies the query. For instance, the attributes of the project relation are given by P, which stands for "Print."

EMP	ENO	ENAME	TITLE
	E2	P.	

ASG	ENO	PNO	RESP	DUR
	E2	P3		

PROJ	PNO	PNAME	BUDGET
	P3	CAD/CAM	

Fig. 2.12 Retrieval Query in QBE

By default, all queries are retrieval. An update query requires the specification of U under the name of the updated relation or in the updated column. The retrieval query corresponding to Example 2.12 is given in Figure 2.12 and the update query of Example 2.13 is given in Figure 2.13. To distinguish examples from constants, examples are underlined.

PAY	TITLE	SAL
	Programmer	U.25000

Fig. 2.13 Update Query in QBE

2.2 Review of Computer Networks

In this section we discuss computer networking concepts relevant to distributed database systems. We omit most of the details of the technological and technical issues in favor of discussing the main concepts.

We define a *computer network* as an *interconnected collection of autonomous computers that are capable of exchanging information among themselves* (Figure 2.14). The keywords in this definition are *interconnected* and *autonomous*. We want the computers to be autonomous so that each computer can execute programs on its own. We also want the computers to be interconnected so that they are capable of exchanging information. Computers on a network are referred to as *nodes*, *hosts*, *end systems*, or *sites*. Note that sometimes the terms *host* and *end system* are used to refer

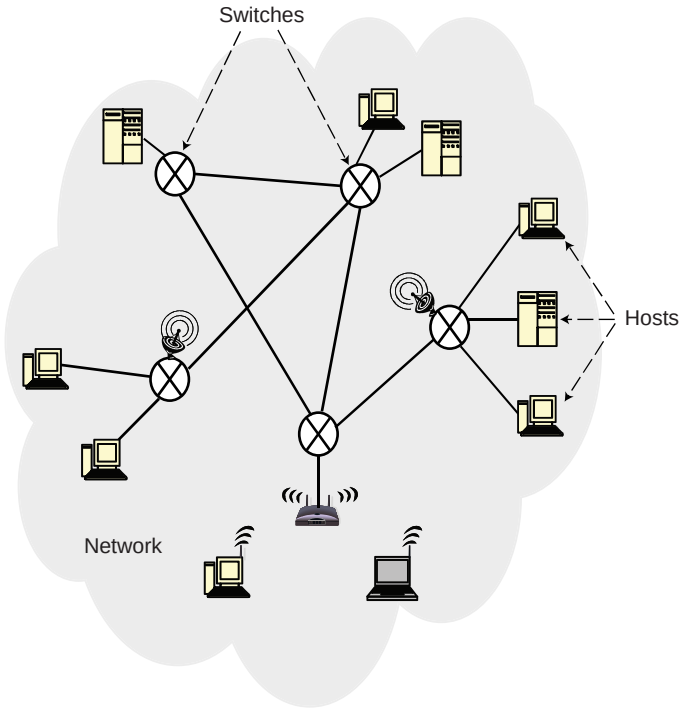

Fig. 2.14 A Computer Network

simply to the equipment, whereas *site* is reserved for the equipment as well as the software that runs on it. Similarly, *node* is generally used as a generic reference to the computers or to the switches in a network. They form one of the fundamental hardware components of a network. The other fundamental component is special purpose devices and links that form the communication path that interconnects the nodes. As depicted in Figure 2.14, the hosts are connected to the network through switches (represented as circles with an X in them)[2], which are special-purpose equipment that *route* messages through the network. Some of the hosts may be connected to the switches directly (using fiber optic, coaxial cable or copper wire) and some via wireless base stations. The switches are connected to each other by communication links that may be fiber optics, coaxial cable, satellite links, microwave connections, etc.

The most widely used computer network these days is the Internet. It is hard to define the Internet since the term is used to mean different things, but perhaps the best definition is that it is a network of networks (Figure 2.15). Each of these

[2] Note that the terms "switch" and "router" are sometimes used interchangeably (even within the same text). However, other times they are used to mean slightly different things: switch refers to the devices inside a network whereas router refers to one that is at the edge of a network connecting it to the backbone. We use them interchangeably as in Figures 2.14 and 2.15.

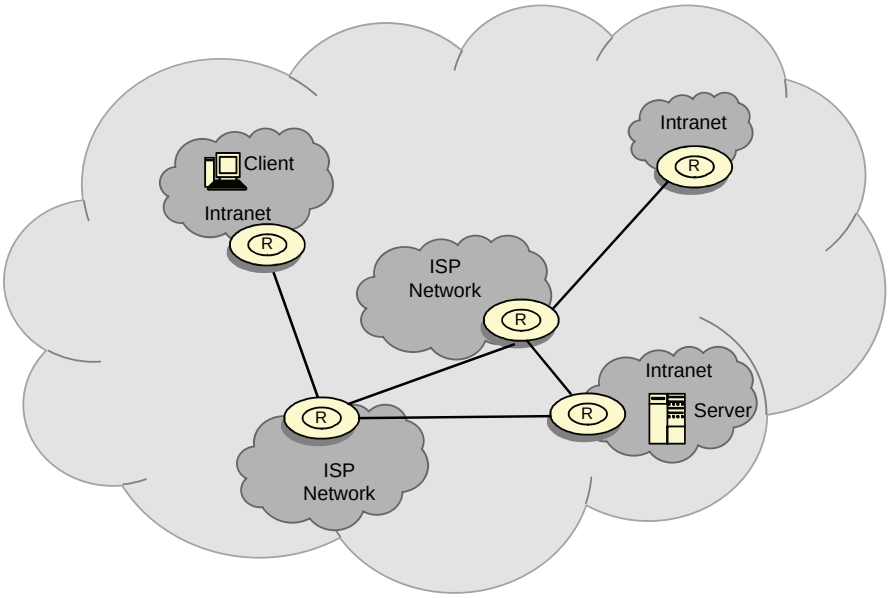

Fig. 2.15 Internet

networks is referred to as an *intranet* to highlight the fact that they are "internal" to an organization. An intranet, then, consists of a set of links and routers (shown as "R" in Figure 2.15) administered by a single administrative entity or by its delegates. For instance, the routers and links at a university constitute a single administrative domain. Such domains may be located within a single geographical area (such as the university network mentioned above), or, as in the case of large enterprises or Internet Service Provider (ISP) networks, span multiple geographical areas. Each intranet is connected to some others by means of links provisioned from ISPs. These links are typically high-speed, long-distance duplex data transmission media (we will define these terms shortly), such as a fiber-optic cable, or a satellite link. These links make up what is called the Internet backbone. Each intranet has a router interface that connects it to the backbone, as shown in Figure 2.15. Thus, each link connects an intranet router to an ISP's router. ISP's routers are connected by similar links to routers of other ISPs. This allows servers and clients within an intranet to communicate with servers and clients in other intranets.

2.2.1 Types of Networks

There are various criteria by which computer networks can be classified. One criterion is the geographic distribution (also called *scale* [Tanenbaum, 2003]), a second

criterion is the *interconnection structure* of nodes (also called *topology*), and the third is the mode of transmission.

2.2.1.1 Scale

In terms of geographic distribution, networks are classified as wide area networks, metropolitan area networks and local area networks. The distinctions among these are somewhat blurred, but in the following, we give some general guidelines that identify each of these networks. The primary distinction among them are probably in terms of propagation delay, administrative control, and the protocols that are used in managing them.

A wide area network (WAN) is one where the link distance between any two nodes is greater than approximately 20 kilometers (km) and can go as large as thousands of kilometers. Use of switches allow the aggregation of communication over wider areas such as this. Owing to the distances that need to be traveled, long delays are involved in wide area data transmission. For example, via satellite, there is a minimum delay of half a second for data to be transmitted from the source to the destination and acknowledged. This is because the speed with which signals can be transmitted is limited to the speed of light, and the distances that need to be spanned are great (about 31,000 km from an earth station to a satellite).

WANs are typically characterized by the heterogeneity of the transmission media, the computers, and the user community involved. Early WANs had a limited capacity of less than a few megabits-per-second (Mbps). However, most of the current ones are broadband WANs that provide capacities of 150 Mbps and above. These individual channels are aggregated into the backbone links; the current backbone links are commonly OC48 at 2.4 Gbps or OC192 at 10Gbps. These networks can carry multiple data streams with varying characteristics (e.g., data as well as audio/video streams), the possibility of negotiating for a level of quality of service (QoS) and reserving network resources sufficient to fulfill this level of QoS.

Local area networks (LANs) are typically limited in geographic scope (usually less than 2 km). They provide higher capacity communication over inexpensive transmission media. The capacities are typically in the range of 10-1000 Mbps per connection. Higher capacity and shorter distances between hosts result in very short delays. Furthermore, the better controlled environments in which the communication links are laid out (within buildings, for example) reduce the noise and interference, and the heterogeneity among the computers that are connected is easier to manage, and a common transmission medium is used.

Metropolitan area networks (MANs) are in between LANs and WANs in scale and cover a city or a portion of it. The distances between nodes is typically on the order of 10 km.

2.2.1.2 Topology

As the name indicates, interconnection structure or topology refers to the way nodes on a network are interconnected. The network in Figure 2.14 is what is called an *irregular* network, where the interconnections between nodes do not follow any pattern. It is possible to find a node that is connected to only one other node, as well as nodes that have connections to a number of nodes. Internet is a typical irregular network.

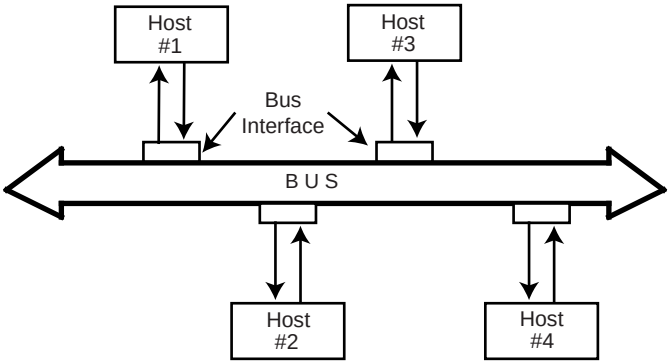

Fig. 2.16 Bus Network

Another popular topology is the bus, where all the computers are connected to a common channel (Figure 2.16). This type of network is primarily used in LANs. The link control is typically performed using *carrier sense medium access with collision detection* (CSMA/CD) protocol. The CSMA/CD bus control mechanism can best be described as a "listen before and while you transmit" scheme. The fundamental point is that each host listens continuously to what occurs on the bus. When a message transmission is detected, the host checks if the message is addressed to it, and takes the appropriate action. If it wants to transmit, it waits until it detects no more activity on the bus and then places its message on the network and continues to listen to bus activity. If it detects another transmission while it is transmitting a message itself, then there has been a "collision." In such a case, and when the collision is detected, the transmitting hosts abort the transmission, each waits a random amount of time, and then each retransmits the message. The basic CSMA/CD scheme is used in the Ethernet local area network[3].

Other common alternatives are star, ring, bus, and mesh networks.

[3] In most current implementations of Ethernet, multiple busses are linked via one or more switches (called *switched hubs*) for expanded coverage and to better control the load on each bus segment. In these systems, individual computers can directly be connected to the switch as well. These are known as switched Ethernet.

- *Star* networks connect all the hosts to a central node that coordinates the transmission on the network. Thus if two hosts want to communicate, they have to go through the central node. Since there is a separate link between the central node and each of the others, there is a negotiation between the hosts and the central node when they wish to communicate.
- *Ring* networks interconnect the hosts in the form of a loop. This type of network was originally proposed for LANs, but their use in these networks has nearly stopped. They are now primarily used in MANs (e.g., SONET rings). In their current incarnation, data transmission around the ring is usually bidirectional (original rings were unidirectional), with each station (actually the interface to which each station is connected) serving as an active repeater that receives a message, checks the address, copies the message if it is addressed to that station, and retransmits it.
  Control of communication in ring type networks is generally controlled by means of a *control token*. In the simplest type of token ring networks, a token, which has one bit pattern to indicate that the network is free and a different bit pattern to indicate that it is in use, is circulated around the network. Any site wanting to transmit a message waits for the token. When it arrives, the site checks the token's bit pattern to see if the network is free or in use. If it is free, the site changes the bit pattern to indicate that the network is in use and then places the messages on the ring. The message circulates around the ring and returns to the sender which changes the bit pattern to free and sends the token to the next computer down the line.
- *Complete* (or *mesh*) interconnection is one where each node is interconnected to every other node. Such an interconnection structure obviously provides more reliability and the possibility of better performance than that of the structures noted previously. However, it is also the costliest. For example, a complete connection of 10,000 computers would require approximately $(10,000)^2$ links.[4]

2.2.2 Communication Schemes

In terms of the physical communication schemes employed, networks can be either *point-to-point* (also called *unicast*) networks, or *broadcast* (sometimes also called *multi-point*) networks.

In point-to-point networks, there are one or more (direct or indirect) links between each pair of nodes. The communication is always between two nodes and the receiver and sender are identified by their addresses that are included in the message header. Data transmission from the sender to the receiver follows one of the possibly many links between them, some of which may involve visiting other intermediate nodes. An intermediate node checks the destination address in the message header and if it is not addressed to it, passes it along to the next intermediate node. This is the

[4] The general form of the equation is $n(n-1)/2$, where n is the number of nodes on the network.

process of *switching* or *routing*. The selection of the links via which messages are sent is determined by usually elaborate routing algorithms that are beyond our scope. We discuss the details of switching in Section 2.2.3.

The fundamental transmission media for point-to-point networks are twisted pair, coaxial or fiber optic cables. Each of these media have different capacities: twisted pair 300 bps to 10 Mbps, coaxial up to 200 Mbps, and fiber optic 10 Gbps and even higher.

In broadcast networks, there is a common communication channel that is utilized by all the nodes in the network. Messages are transmitted over this common channel and received by all the nodes. Each node checks the receiver address and if the message is not addressed to it, ignores it.

A special case of broadcasting is *multicasting* where the message is sent to a subset of the nodes in the network. The receiver address is somehow encoded to indicate which nodes are the recipients.

Broadcast networks are generally radio or satellite-based. In case of satellite transmission, each site beams its transmission to a satellite which then beams it back at a different frequency. Every site on the network listens to the receiving frequency and has to disregard the message if it is not addressed to that site. A network that uses this technique is HughesNet^{TM}.

Microwave transmission is another mode of data communication and it can be over satellite or terrestrial. Terrestrial microwave links used to form a major portion of most countries' telephone networks although many of these have since been converted to fiber optic. In addition to the public carriers, some companies make use of private terrestrial microwave links. In fact, major metropolitan cities face the problem of microwave interference among privately owned and public carrier links. A very early example that is usually identified as having pioneered the use of satellite microwave transmission is ALOHA [Abramson, 1973].

Satellite and microwave networks are examples of wireless networks. These types of wireless networks are commonly referred to as *wireless broadband* networks. Another type of wireless network is one that is based on *cellular* networks. A cellular network control station is responsible for a geographic area called a *cell* and coordinates the communication from mobile hosts in their cell. These control stations may be linked to a "wireline" backbone network and thereby provide access from/to mobile hosts to other mobile hosts or stationary hosts on the wireline network.

A third type of wireless network with which most of us may be more familiar are *wireless LANs* (commonly referred to as Wi-LAN or WiLan). In this case a number of "base stations" are connected to a wireline network and serve as connection points for mobile hosts (similar to control stations in cellular networks). These networks can provide bandwidth of up to 54 Mbps.

A final word on broadcasting topologies is that they have the advantage that it is easier to check for errors and to send messages to more than one site than to do so in point-to-point topologies. On the other hand, since everybody listens in, broadcast networks are not as secure as point-to-point networks.

2.2.3 Data Communication Concepts

What we refer to as data communication is the set of technologies that enable two hosts to communicate. We are not going to be too detailed in this discussion, since, at the distributed DBMS level, we can assume that the technology exists to move bits between hosts. We, instead, focus on a few important issues that are relevant to understanding delay and routing concepts.

As indicated earlier hosts are connected by *links*, each of which can carry one or more *channels*. Link is a physical entity whereas channel is a logical one. Communication links can carry signals either in digital form or in analog form. Telephone lines, for example, can carry data in analog form between the home and the central office – the rest of the telephone network is now digital and even the home-to-central office link is becoming digital with voice-over-IP (VoIP) technology. Each communication channel has a *capacity*, which can be defined as the amount of information that can be transmitted over the channel in a given time unit. This capacity is commonly referred to as the *bandwidth* of the channel. In analog transmission channels, the bandwidth is defined as the difference (in hertz) between the lowest and highest frequencies that can be transmitted over the channel per second. In digital links, *bandwidth* refers (less formally and with abuse of terminology) to the number of bits that can be transmitted per second (bps).

With respect to delays in getting the user's work done, the bandwidth of a transmission channel is a significant factor, but it is not necessarily the only ones. The other factor in the transmission time is the software employed. There are usually overhead costs involved in data transmission due to the redundancies within the message itself, necessary for error detection and correction. Furthermore, the network software adds headers and trailers to any message, for example, to specify the destination or to check for errors in the entire message. All of these activities contribute to delays in transmitting data. The actual rate at which data are transmitted across the network is known as the *data transfer rate* and this rate is usually less than the actual bandwidth of the transmission channel. The software issues, that generally are referred as *network protocols*, are discussed in the next section.

In computer-to-computer communication, data are usually transmitted in *packets*, as we mentioned earlier. Usually, upper limits on frame sizes are established for each network and each contains data as well as some control information, such as the destination and source addresses, block error check codes, and so on (Figure 2.17). If a message that is to be sent from a source node to a destination node cannot fit one frame, it is split over a number of frames. This is be discussed further in Section 2.2.4.

There are various possible forms of switching/routing that can occur in point-to-point networks. It is possible to establish a connection such that a dedicated channel exists between the sender and the receiver. This is called *circuit switching* and is commonly used in traditional telephone connections. When a subscriber dials the number of another subscriber, a circuit is established between the two phones by means of various switches. The circuit is maintained during the period of conversation and is broken when one side hangs up. Similar setup is possible in computer networks.

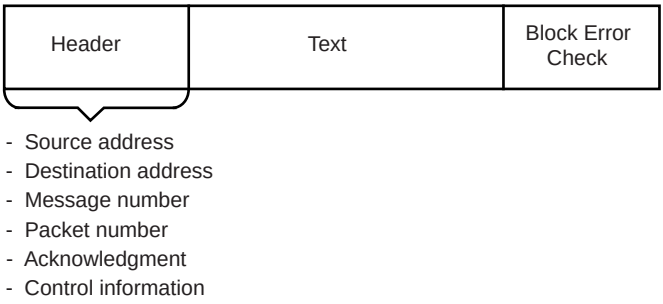

Fig. 2.17 Typical Frame Format

Another form of switching used in computer communication is *packet switching*, where a message is broken up into packets and each packet transmitted individually. In our discussion of the TCP/IP protocol earlier, we referred to messages being transmitted; in fact the TCP protocol (or any other transport layer protocol) takes each application package and breaks it up into fixed sized packets. Therefore, each application message may be sent to the destination as multiple packets.

Packets for the same message may travel independently of each other and may, in fact, take different routes. The result of routing packets along possibly different links in the network is that they may arrive at the destination out-of-order. Thus the transport layer software at the destination site should be able to sort them into their original order to reconstruct the message. Consequently, it is the individual packages that are routed through the network, which may result in packets reaching the destination at different times and even out of order. The transport layer protocol at the destination is responsible for collating and ordering the packets and generating the application message properly.

The advantages of packet switching are many. First, packet-switching networks provide higher link utilization since each link is not dedicated to a pair of communicating equipment and can be shared by many. This is especially useful in computer communication due to its bursty nature – there is a burst of transmission and then some break before another burst of transmission starts. The link can be used for other transmission when it is idle. Another reason is that packetizing may permit the parallel transmission of data. There is usually no requirement that various packets belonging to the same message travel the same route through the network. In such a case, they may be sent in parallel via different routes to improve the total data transmission time. As mentioned above, the result of routing frames this way is that their in-order delivery cannot be guaranteed.

On the other hand, circuit switching provides a dedicated channel between the receiver and the sender. If there is a sizable amount of data to be transmitted between the two or if the channel sharing in packet switched networks introduces too much delay or delay variance, or packet loss (which are important in multimedia applications), then the dedicated channel facilitates this significantly. Therefore, schemes similar to circuit switching (i.e., reservation-based schemes) have gained favor in

the broadband networks that support applications such as multimedia with very high data transmission loads.

2.2.4 Communication Protocols

Establishing a physical connection between two hosts is not sufficient for them to communicate. Error-free, reliable and efficient communication between hosts requires the implementation of elaborate software systems that are generally called *protocols*. Network protocols are "layered" in that network functionality is divided into layers, each layer performing a well-defined function relying on the services provided by the layer below it and providing a service to the layer above. A protocol defines the services that are performed at one layer. The resulting layered protocol set is referred to as a *protocol stack* or *protocol suite*.

There are different protocol stacks for different types of networks; however, for communication over the Internet, the standard one is what is referred to as TCP/IP that stands for "Transport Control Protocol/Internet Protocol". We focus primarily on TCP/IP in this section as well as some of the common LAN protocols.

Before we get into the specifics of the TCP/IP protocol stack, let us first discuss how a message from a process on host C in Figure 2.15 is transmitted to a process on server S, assuming both hosts implement the TCP/IP protocol. The process is depicted in Figure 2.18.

The appropriate application layer protocol takes the message from the process on host C and creates an application layer message by adding some application layer header information (oblique hatched part in Figure 2.18) details of which are not important for us. The application message is handed over to the TCP protocol, which repeats the process by adding its own header information. TCP header includes the necessary information to facilitate the provision of TCP services we discuss shortly. The Internet layer takes the TCP message that is generated and forms an Internet message as we also discuss below. This message is now physically transmitted from host C to its router using the protocol of its own network, then through a series of routers to the router of the network that contains server S, where the process is reversed until the original message is recovered and handed over to the appropriate process on S. The TCP protocols at hosts C and S communicate to ensure the end-to-end guarantees that we discussed.

2.2.4.1 TCP/IP Protocol Stack

What is referred to as TCP/IP is in fact a family of protocols, commonly referred to as the *protocol stack*. It consists of two sets of protocols, one set at the *transport layer* and the other at the *network (Internet) layer* (Figure 2.19).

The transport layer defines the types of services that the network provides to applications. The protocols at this layer address issues such as data loss (can the

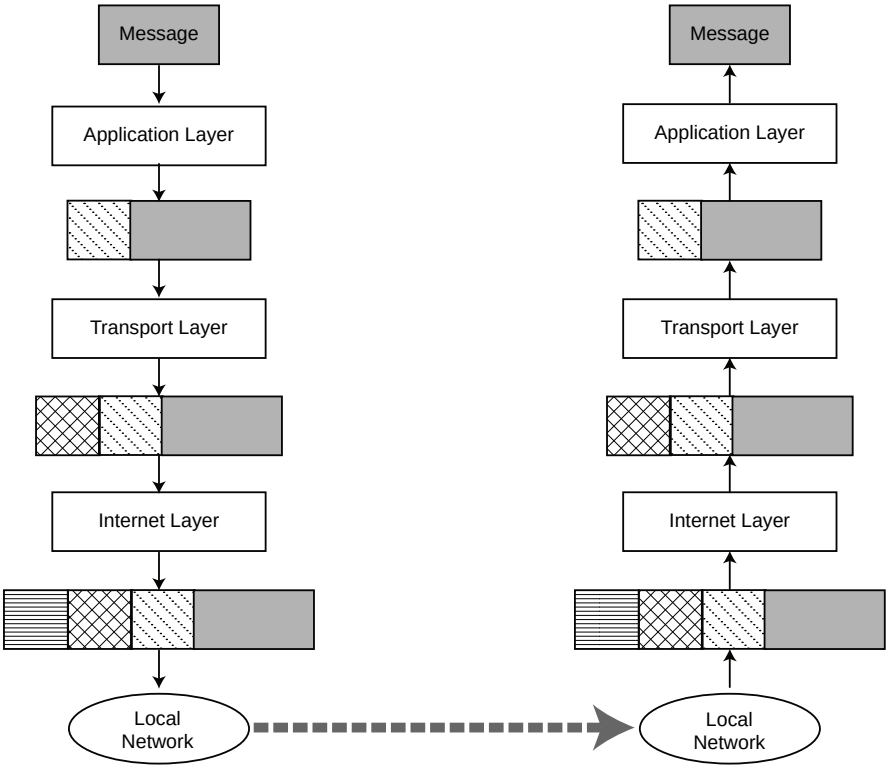

Fig. 2.18 Message Transmission using TCP/IP

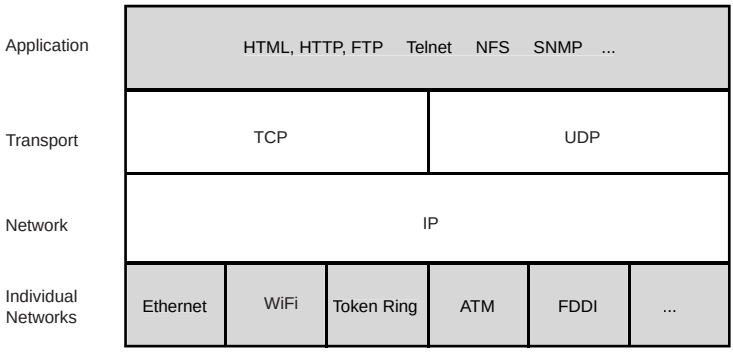

Fig. 2.19 TCP/IP Protocol

application tolerate losing some of the data during transmission?), bandwidth (some applications have minimum bandwidth requirements while others can be more elastic in their requirements), and timing (what type of delay can the applications tolerate?). For example, a file transfer application can not tolerate any data loss, can be flexible in its bandwidth use (it will work whether the connection is high capacity or low capacity, although the performance may differ), and it does not have strict timing requirements (although we may not like a file transfer to take a few days, it would still work). In contrast, a real-time audio/video transmission application can tolerate a limited amount of data loss (this may cause some jitter and other problems, but the communication will still be "understandable"), has minimum bandwidth requirement (5-128 Kbps for audio and 5 Kbps-20 Mbps for video), and is time sensitive (audio and video data need to be synchronized).

To deal with these varying requirements (at least with some of them), at the transport layer, two protocols are provided: TCP and UDP. TCP is connection-oriented, meaning that prior setup is required between the sender and the receiver before actual message transmission can start; it provides reliable transmission between the sender and the receiver by ensuring that the messages are received correctly at the receiver (referred to as "end-to-end reliability"); ensures flow control so that the sender does not overwhelm the receiver if the receiver process is not able to keep up with the incoming messages, and ensures congestion control so that the sender is throttled when network is overloaded. Note that TCP does not address the timing and minimum bandwidth guarantees, leaving these to the application layer.

UDP, on the other hand, is a connectionless service that does not provide the reliability, flow control and congestion control guarantees that TCP provides. Nor does it establish a connection between the sender and receiver beforehand. Thus, each message is transmitted hoping that it will get to the destination, but no end-to-end guarantees are provided. Thus, UDP has significantly lower overhead than TCP, and is preferred by applications that would prefer to deal with these requirements themselves, rather than having the network protocol handle them.

The network layer implements the Internet Protocol (IP) that provides the facility to "package" a message in a standard Internet message format for transmission across the network. Each Internet message can be up to 64KB long and consists of a header that contains, among other things, the IP addresses of the sender and the receiver machines (the numbers such as 129.97.79.58 that you may have seen attached to your own machines), and the message body itself. The message format of each network that makes up the Internet can be different, but each of these messages are encoded into an Internet message by the Internet Protocol before they are transmitted[5].

The importance of TCP/IP is the following. Each of the intranets that are part of the Internet can use its own preferred protocol, so the computers on that network implement that particular protocol (e.g., the token ring mechanism and the CSMA/CS technique described above are examples of these types of protocols). However, if they are to connect to the Internet, they need to be able to communicate using TCP/IP, which are implemented on top of these specific network protocols (Figure 2.19).

[5] Today, many of the Intranets also use TCP/IP, in which case IP encapsulation may not be necessary.

2.2.4.2 Other Protocol Layers

Let us now briefly consider the other two layers depicted in Figure 2.19. Although these are not part of the TCP/IP protocol stack, they are necessary to be able to build distributed applications. These make up the top and the bottom layers of the protocol stack.

The Application Protocol layer provides the specifications that distributed applications have to follow. For example, if one is building a Web application, then the documents that will be posted on the Web have to be written according to the HTML protocol (note that HTML is not a networking protocol, but a document encoding protocol) and the communication between the client browser and the Web server has to follow the HTTP protocol. Similar protocols are defined at this layer for other applications as indicated in the figure.

The bottom layer represents the specific network that may be used. Each of those networks have their own message formats and protocols and they provide the mechanisms for data transmission within those networks.

The standardization for LANs is spearheaded by the Institute of Electrical and Electronics Engineers (IEEE), specifically their Committee No. 802; hence the standard that has been developed is known as the IEEE 802 Standard. The three layers of the IEEE 802 local area network standard are the physical layer, the medium access control layer, and the logical link control layer.

The physical layer deals with physical data transmission issues such as signaling. Medium access control layer defines protocols that control who can have access to the transmission medium and when. Logical link control layer implements protocols that ensure reliable packet transmission between two adjacent computers (not end-to-end). In most LANs, the TCP and IP layer protocols are implemented on top of these three layers, enabling each computer to be able to directly communicate on the Internet.

To enable it to cover a variety of LAN architectures, the 802 local area network standard is actually a number of standards rather than a single one. Originally, it was specified to support three mechanisms at the medium access control level: the CSMA/CD mechanism, token ring, and token access mechanism for bus networks.

2.3 Bibliographic Notes

This chapter covered the basic issues related to relational database systems and computer networks. These concepts are discussed in much greater detail in a number of excellent textbooks. Related to database technology, we can name [Ramakrishnan and Gehrke, 2003; Elmasri and Navathe, 2011; Silberschatz et al., 2002; Garcia-Molina et al., 2002; Kifer et al., 2006], and [Date, 2004]. For computer networks one can refer to [Tanenbaum, 2003; Kurose and Ross, 2010; Leon-Garcia and Widjaja, 2004; Comer, 2009]. More focused discussion of data communication issues can be found in [Stallings, 2011].

Chapter 3
Distributed Database Design

The design of a distributed computer system involves making decisions on the placement of *data* and *programs* across the sites of a computer network, as well as possibly designing the network itself. In the case of distributed DBMSs, the distribution of applications involves two things: the distribution of the distributed DBMS software and the distribution of the application programs that run on it. Different architectural models discussed in Chapter 1 address the issue of application distribution. In this chapter we concentrate on distribution of data.

It has been suggested that the organization of distributed systems can be investigated along three orthogonal dimensions [Levin and Morgan, 1975] (Figure 3.1):

1. Level of sharing
2. Behavior of access patterns
3. Level of knowledge on access pattern behavior

In terms of the level of sharing, there are three possibilities. First, there is *no sharing*: each application and its data execute at one site, and there is no communication with any other program or access to any data file at other sites. This characterizes the very early days of networking and is probably not very common today. We then find the level of *data sharing*; all the programs are replicated at all the sites, but data files are not. Accordingly, user requests are handled at the site where they originate and the necessary data files are moved around the network. Finally, in *data-plus-program sharing*, both data and programs may be shared, meaning that a program at a given site can request a service from another program at a second site, which, in turn, may have to access a data file located at a third site.

Levin and Morgan draw a distinction between data sharing and data-plus-program sharing to illustrate the differences between homogeneous and heterogeneous distributed computer systems. They indicate, correctly, that in a heterogeneous environment it is usually very difficult, and sometimes impossible, to execute a given program on different hardware under a different operating system. It might, however, be possible to move data around relatively easily.

M.T. Özsu and P. Valduriez, *Principles of Distributed Database Systems: Third Edition*,
DOI 10.1007/978-1-4419-8834-8_3,

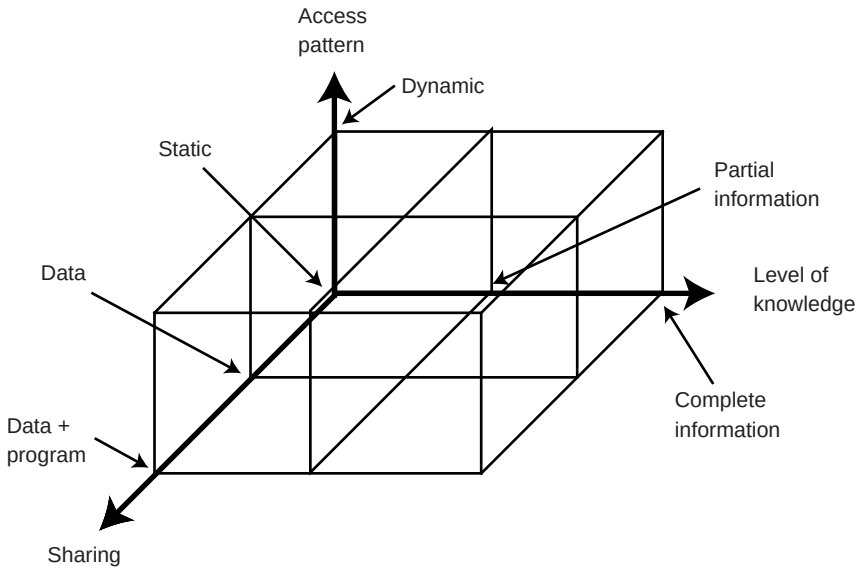

Fig. 3.1 Framework of Distribution

Along the second dimension of access pattern behavior, it is possible to identify two alternatives. The access patterns of user requests may be *static*, so that they do not change over time, or *dynamic*. It is obviously considerably easier to plan for and manage the static environments than would be the case for dynamic distributed systems. Unfortunately, it is difficult to find many real-life distributed applications that would be classified as static. The significant question, then, is not whether a system is static or dynamic, but how dynamic it is. Incidentally, it is along this dimension that the relationship between the distributed database design and query processing is established (refer to Figure 1.7).

The third dimension of classification is the level of knowledge about the access pattern behavior. One possibility, of course, is that the designers do not have any information about how users will access the database. This is a theoretical possibility, but it is very difficult, if not impossible, to design a distributed DBMS that can effectively cope with this situation. The more practical alternatives are that the designers have *complete information*, where the access patterns can reasonably be predicted and do not deviate significantly from these predictions, or *partial information*, where there are deviations from the predictions.

The distributed database design problem should be considered within this general framework. In all the cases discussed, except in the no-sharing alternative, new problems are introduced in the distributed environment which are not relevant in a centralized setting. In this chapter it is our objective to focus on these unique problems.

Two major strategies that have been identified for designing distributed databases are the *top-down approach* and the *bottom-up approach* [Ceri et al., 1987]. As the names indicate, they constitute very different approaches to the design process. Top-down approach is more suitable for tightly integrated, homogeneous distributed DBMSs, while bottom-up design is more suited to multidatabases (see the classification in Chapter 1). In this chapter, we focus on top-down design and defer bottom-up to the next chapter.

3.1 Top-Down Design Process

A framework for top-down design process is shown in Figure 3.2. The activity begins with a requirements analysis that defines the environment of the system and "elicits both the data and processing needs of all potential database users" [Yao et al., 1982a]. The requirements study also specifies where the final system is expected to stand with respect to the objectives of a distributed DBMS as identified in Section 1.4. These objectives are defined with respect to performance, reliability and availability, economics, and expandability (flexibility).

The requirements document is input to two parallel activities: view design and conceptual design. The *view design* activity deals with defining the interfaces for end users. The *conceptual design*, on the other hand, is the process by which the enterprise is examined to determine entity types and relationships among these entities. One can possibly divide this process into two related activity groups [Davenport, 1981]: entity analysis and functional analysis. *Entity analysis* is concerned with determining the entities, their attributes, and the relationships among them. *Functional analysis*, on the other hand, is concerned with determining the fundamental functions with which the modeled enterprise is involved. The results of these two steps need to be cross-referenced to get a better understanding of which functions deal with which entities.

There is a relationship between the conceptual design and the view design. In one sense, the conceptual design can be interpreted as being an integration of user views. Even though this *view integration* activity is very important, the conceptual model should support not only the existing applications, but also future applications. View integration should be used to ensure that entity and relationship requirements for all the views are covered in the conceptual schema.

In conceptual design and view design activities the user needs to specify the data entities and must determine the applications that will run on the database as well as statistical information about these applications. Statistical information includes the specification of the frequency of user applications, the volume of various information, and the like. Note that from the conceptual design step comes the definition of global conceptual schema discussed in Section 1.7. We have not yet considered the implications of the distributed environment; in fact, up to this point, the process is identical to that in a centralized database design.

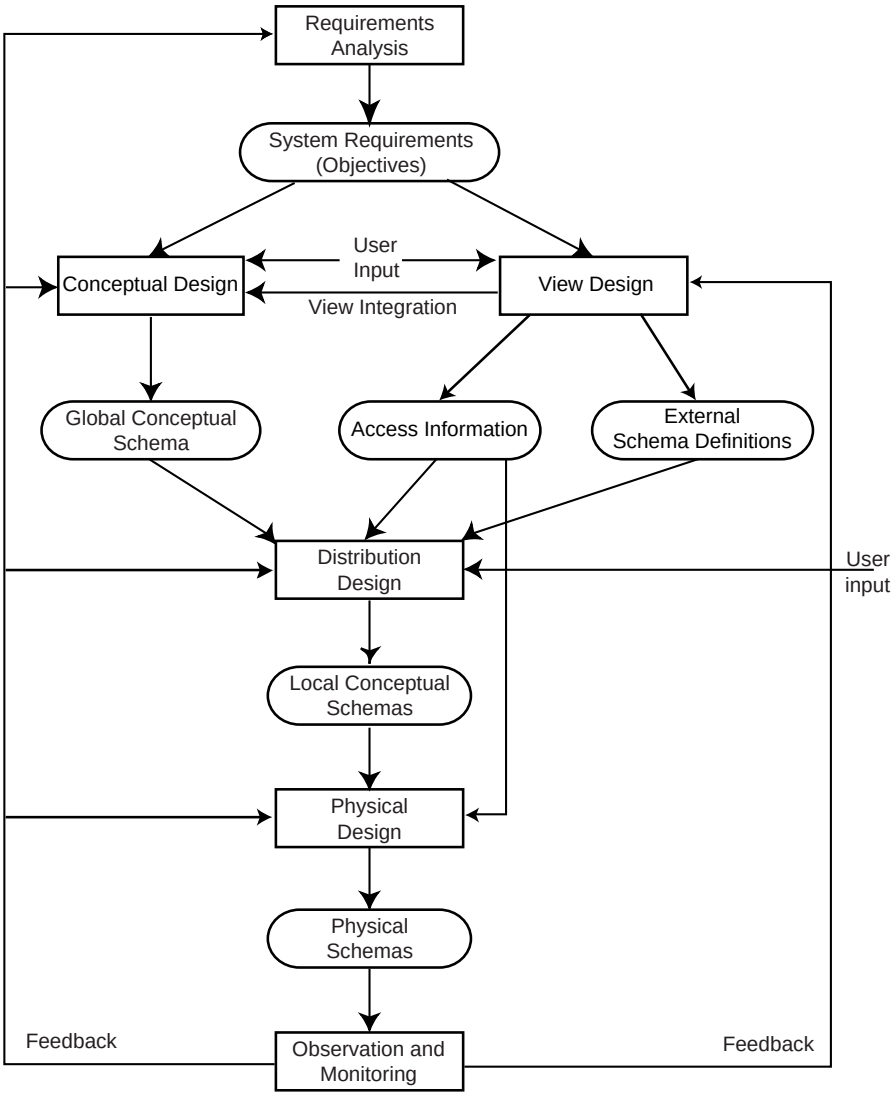

Fig. 3.2 Top-Down Design Process

The global conceptual schema (GCS) and access pattern information collected as a result of view design are inputs to the *distribution design* step. The objective at this stage, which is the focus of this chapter, is to design the local conceptual schemas (LCSs) by distributing the entities over the sites of the distributed system. It is possible, of course, to treat each entity as a unit of distribution. Given that we use

the relational model as the basis of discussion in this book, the entities correspond to relations.

Rather than distributing relations, it is quite common to divide them into subrelations, called *fragments*, which are then distributed. Thus, the distribution design activity consists of two steps: *fragmentation* and *allocation*. The reason for separating the distribution design into two steps is to better deal with the complexity of the problem. However, this raises other concerns as we discuss at the end of the chapter.

The last step in the design process is the physical design, which maps the local conceptual schemas to the physical storage devices available at the corresponding sites. The inputs to this process are the local conceptual schema and the access pattern information about the fragments in them.

It is well known that design and development activity of any kind is an ongoing process requiring constant monitoring and periodic adjustment and tuning. We have therefore included observation and monitoring as a major activity in this process. Note that one does not monitor only the behavior of the database implementation but also the suitability of user views. The result is some form of feedback, which may result in backing up to one of the earlier steps in the design.

3.2 Distribution Design Issues

In the preceding section we indicated that the relations in a database schema are usually decomposed into smaller fragments, but we did not offer any justification or details for this process. The objective of this section is to fill in these details.

The following set of interrelated questions covers the entire issue. We will therefore seek to answer them in the remainder of this section.

1. Why fragment at all?
2. How should we fragment?
3. How much should we fragment?
4. Is there any way to test the correctness of decomposition?
5. How should we allocate?
6. What is the necessary information for fragmentation and allocation?

3.2.1 Reasons for Fragmentation

From a data distribution viewpoint, there is really no reason to fragment data. After all, in distributed file systems, the distribution is performed on the basis of entire files. In fact, the very early work dealt specifically with the allocation of files to nodes on a computer network. We consider earlier models in Section 3.4.

With respect to fragmentation, the important issue is the appropriate unit of distribution. A relation is not a suitable unit, for a number of reasons. First, application views are usually subsets of relations. Therefore, the locality of accesses of applications is defined not on entire relations but on their subsets. Hence it is only natural to consider subsets of relations as distribution units.

Second, if the applications that have views defined on a given relation reside at different sites, two alternatives can be followed, with the entire relation being the unit of distribution. Either the relation is not replicated and is stored at only one site, or it is replicated at all or some of the sites where the applications reside. The former results in an unnecessarily high volume of remote data accesses. The latter, on the other hand, has unnecessary replication, which causes problems in executing updates (to be discussed later) and may not be desirable if storage is limited.

Finally, the decomposition of a relation into fragments, each being treated as a unit, permits a number of transactions to execute concurrently. In addition, the fragmentation of relations typically results in the parallel execution of a single query by dividing it into a set of subqueries that operate on fragments. Thus fragmentation typically increases the level of concurrency and therefore the system throughput. This form of concurrency, which we refer to as *intraquery concurrency*, is dealt with mainly in Chapters 7 and 8, under query processing.

Fragmentation raises difficulties as well. If the applications have conflicting requirements that prevent decomposition of the relation into mutually exclusive fragments, those applications whose views are defined on more than one fragment may suffer performance degradation. It might, for example, be necessary to retrieve data from two fragments and then take their join, which is costly. Minimizing distributed joins is a fundamental fragmentation issue.

The second problem is related to semantic data control, specifically to integrity checking. As a result of fragmentation, attributes participating in a dependency may be decomposed into different fragments that might be allocated to different sites. In this case, even the simpler task of checking for dependencies would result in chasing after data in a number of sites. In Chapter 5 we return to the issue of semantic data control.

3.2.2 Fragmentation Alternatives

Relation instances are essentially tables, so the issue is one of finding alternative ways of dividing a table into smaller ones. There are clearly two alternatives for this: dividing it *horizontally* or dividing it *vertically*.

Example 3.1. In this chapter we use a modified version of the relational database scheme developed in Section 2.1. We have added to the PROJ relation a new attribute (LOC) that indicates the place of each project. Figure 3.3 depicts the database instance we will use. Figure 3.4 shows the PROJ relation of Figure 3.3 divided horizontally into two relations. Subrelation PROJ_1 contains information about projects whose

EMP

ENO	ENAME	TITLE
E1	J. Doe	Elect. Eng
E2	M. Smith	Syst. Anal.
E3	A. Lee	Mech. Eng.
E4	J. Miller	Programmer
E5	B. Casey	Syst. Anal.
E6	L. Chu	Elect. Eng.
E7	R. Davis	Mech. Eng.
E8	J. Jones	Syst. Anal.

ASG

ENO	PNO	RESP	DUR
E1	P1	Manager	12
E2	P1	Analyst	24
E2	P2	Analyst	6
E3	P3	Consultant	10
E3	P4	Engineer	48
E4	P2	Programmer	18
E5	P2	Manager	24
E6	P4	Manager	48
E7	P3	Engineer	36
E8	P3	Manager	40

PROJ

PNO	PNAME	BUDGET	LOC
P1	Instrumentation	150000	Montreal
P2	Database Develop.	135000	New York
P3	CAD/CAM	250000	New York
P4	Maintenance	310000	Paris

PAY

TITLE	SAL
Elect. Eng.	40000
Syst. Anal.	34000
Mech. Eng.	27000
Programmer	24000

Fig. 3.3 Modified Example Database

budgets are less than \$200,000, whereas $PROJ_2$ stores information about projects with larger budgets. ♦

Example 3.2. Figure 3.5 shows the PROJ relation of Figure 3.3 partitioned vertically into two subrelations, $PROJ_1$ and $PROJ_2$. $PROJ_1$ contains only the information about project budgets, whereas $PROJ_2$ contains project names and locations. It is important to notice that the primary key to the relation (PNO) is included in both fragments. ♦

The fragmentation may, of course, be nested. If the nestings are of different types, one gets *hybrid fragmentation*. Even though we do not treat hybrid fragmentation as a primitive fragmentation strategy, many real-life partitionings may be hybrid.

3.2.3 Degree of Fragmentation

The extent to which the database should be fragmented is an important decision that affects the performance of query execution. In fact, the issues in Section 3.2.1 concerning the reasons for fragmentation constitute a subset of the answers to the question we are addressing here. The degree of fragmentation goes from one extreme, that is, not to fragment at all, to the other extreme, to fragment to the level of

$PROJ_1$

PNO	PNAME	BUDGET	LOC
P1	Instrumentation	150000	Montreal
P2	Database Develop.	135000	New York

$PROJ_2$

PNO	PNAME	BUDGET	LOC
P3	CAD/CAM	255000	New York
P4	Maintenance	310000	Paris

Fig. 3.4 Example of Horizontal Partitioning

$PROJ_1$

PNO	BUDGET
P1	150000
P2	135000
P3	250000
P4	310000

$PROJ_2$

PNO	PNAME	LOC
P1	Instrumentation	Montreal
P2	Database Develop.	New York
P3	CAD/CAM	New York
P4	Maintenance	Paris

Fig. 3.5 Example of Vertical Partitioning

individual tuples (in the case of horizontal fragmentation) or to the level of individual attributes (in the case of vertical fragmentation).

We have already addressed the adverse effects of very large and very small units of fragmentation. What we need, then, is to find a suitable level of fragmentation that is a compromise between the two extremes. Such a level can only be defined with respect to the applications that will run on the database. The issue is, how? In general, the applications need to be characterized with respect to a number of parameters. According to the values of these parameters, individual fragments can be identified. In Section 3.3 we describe how this characterization can be carried out for alternative fragmentations.

3.2.4 Correctness Rules of Fragmentation

We will enforce the following three rules during fragmentation, which, together, ensure that the database does not undergo semantic change during fragmentation.

1. *Completeness*. If a relation instance R is decomposed into fragments $F_R = \{R_1, R_2, \ldots, R_n\}$, each data item that can be found in R can also be found in one or more of R_i's. This property, which is identical to the *lossless decomposition* property of normalization (Section 2.1), is also important in fragmentation since it ensures that the data in a global relation are mapped into fragments without any loss [Grant, 1984]. Note that in the case of horizontal fragmentation, the "item" typically refers to a tuple, while in the case of vertical fragmentation, it refers to an attribute.
2. *Reconstruction*. If a relation R is decomposed into fragments $F_R = \{R_1, R_2, \ldots, R_n\}$, it should be possible to define a relational operator $\bigtriangledown$ such that

   $$R = \bigtriangledown R_i, \quad \forall R_i \in F_R$$

   The operator $\bigtriangledown$ will be different for different forms of fragmentation; it is important, however, that it can be identified. The reconstructability of the relation from its fragments ensures that constraints defined on the data in the form of dependencies are preserved.
3. *Disjointness*. If a relation R is horizontally decomposed into fragments $F_R = \{R_1, R_2, \ldots, R_n\}$ and data item d_i is in R_j, it is not in any other fragment R_k $(k \neq j)$. This criterion ensures that the horizontal fragments are disjoint. If relation R is vertically decomposed, its primary key attributes are typically repeated in all its fragments (for reconstruction). Therefore, in case of vertical partitioning, disjointness is defined only on the non-primary key attributes of a relation.

3.2.5 Allocation Alternatives

Assuming that the database is fragmented properly, one has to decide on the allocation of the fragments to various sites on the network. When data are allocated, it may either be replicated or maintained as a single copy. The reasons for replication are reliability and efficiency of read-only queries. If there are multiple copies of a data item, there is a good chance that some copy of the data will be accessible somewhere even when system failures occur. Furthermore, read-only queries that access the same data items can be executed in parallel since copies exist on multiple sites. On the other hand, the execution of update queries cause trouble since the system has to ensure that all the copies of the data are updated properly. Hence the decision regarding replication is a trade-off that depends on the ratio of the read-only queries to the