Distributed Database Management Systems

A Practical Approach

Saeed K. Rahimi and Frank S. Haug

WILEY

IEEE

IEEE computer society

WWW. LINK AVAILABLE

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

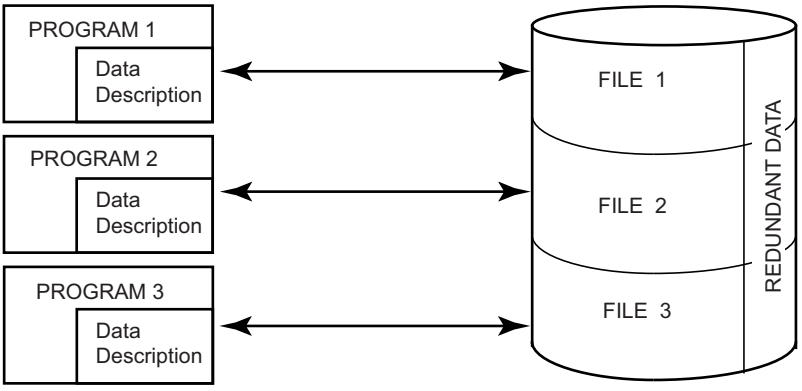


Fig. 1.1 Traditional File Processing

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

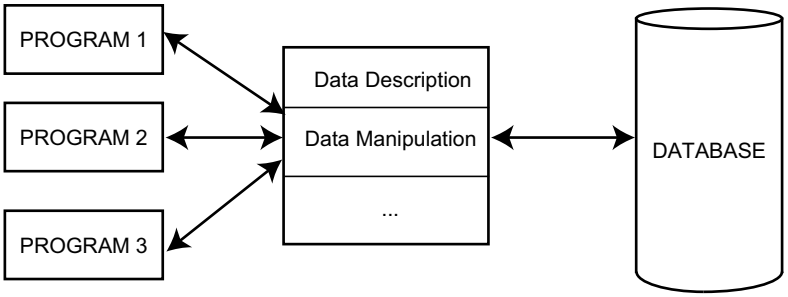


Fig. 1.2 Database Processing

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).

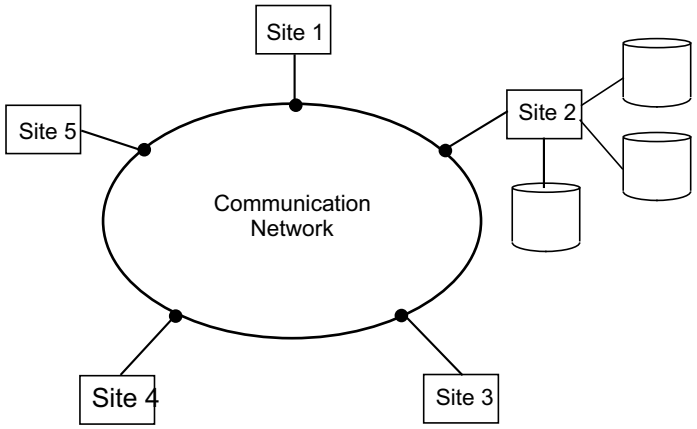


Fig. 1.3 Central Database on a Network

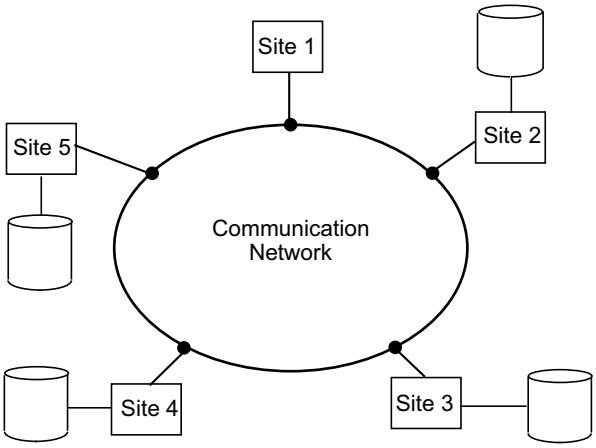


Fig. 1.4 DDBS Environment

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.

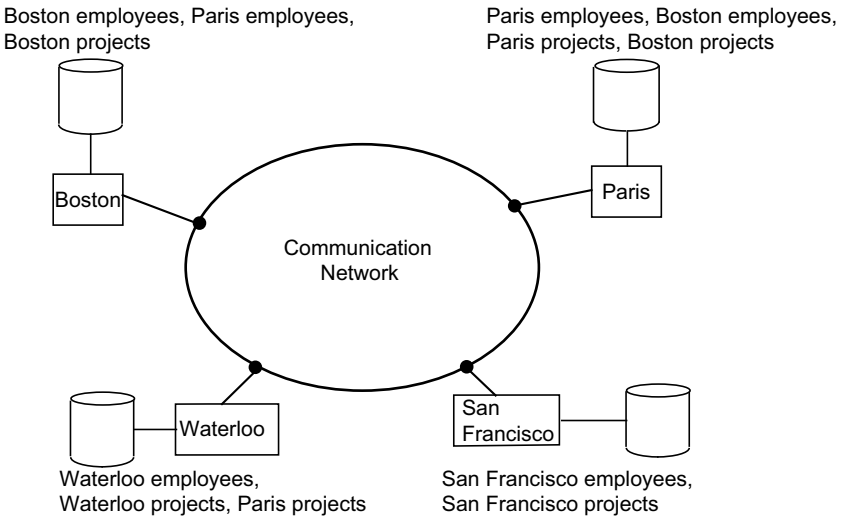


Fig. 1.5 A Distributed Application

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

2

DATA DISTRIBUTION ALTERNATIVES

In a distributed database management system (DDBMS) data is intentionally distributed to take advantage of all computing resources that are available to the organization. For these systems, the schema design is done top–down. A **top–down** design approach considers the data requirements of the **entire organization** and generates a **global conceptual model (GCM)** of all the information that is required. The GCM is then distributed across all appropriate local DBMS (LDBMS) engines to generate the **local conceptual model (LCM)** for each participant LDBMS. As a result, DDBMSs always have **one and only one** GCM and one or more LCM. Figure 2.1 depicts the top–down distribution design approach in a distributed database management system.

By contrast, the design of a federated database system is done from the bottom–up. A **bottom–up** design approach considers the existing data distributed within an organization and uses a process called schema integration to create at least one unified schema (Fig. 2.2). The **unified schema** is similar to the GCM, except that there can be more than one unified schema. **Schema integration** is a process that uses a collection of existing conceptual model elements, which have previously been exported from one or more LCMs, to generate a semantically integrated model (a single, unified schema). We will examine the details of federated database systems in Chapter 12.

Designers of a distributed database (DDB) will decide what distribution alternative is best for a given situation. They may decide to keep every table intact (all rows and all columns of every table are stored in the same DB at the same Site) or to break up some of the tables into smaller chunks of data called **fragments** or **partitions**. In a distributed database, the designers may decide to store these fragments locally (**localized**) or store these fragments across a number of LDBMSs on the network (**distributed**).

Distributed Database Management Systems by Saeed K. Rahimi and Frank S. Haug

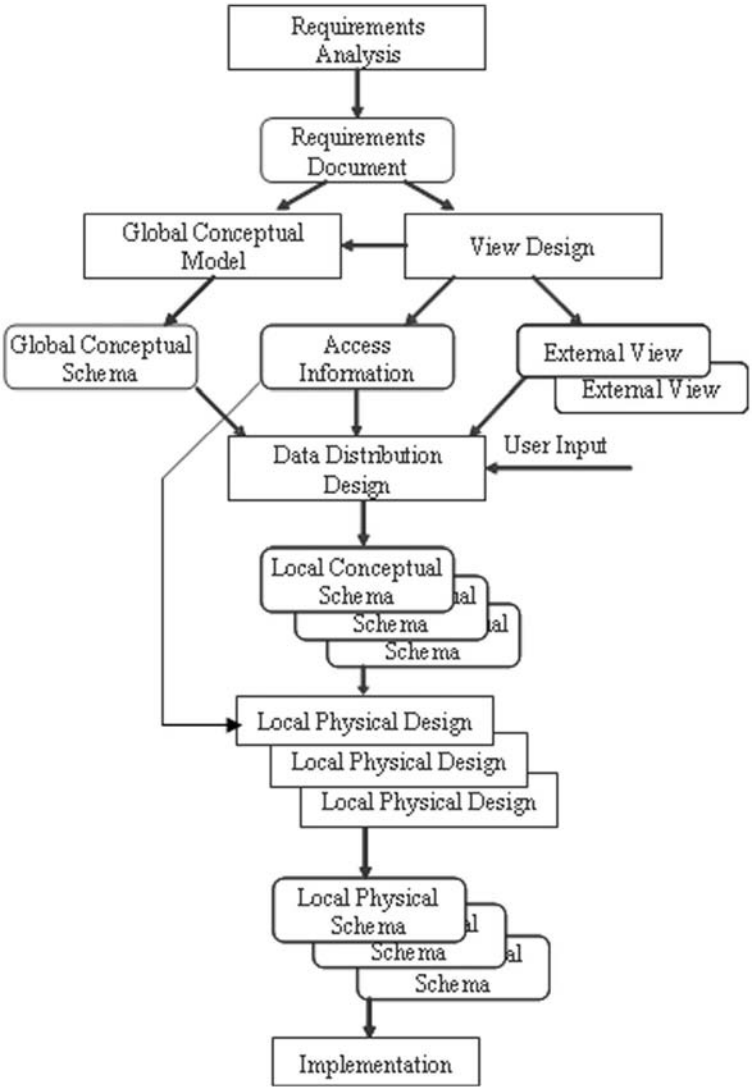


Figure 2.1 The top–down design process for a distributed database system.

Distributed tables can have one of the following forms:

- Nonreplicated, nonfragmented (nonpartitioned)
- Fully replicated (all tables)
- Fragmented (also known as partitioned)
- Partially replicated (some tables or some fragments)
- Mixed (any combination of the above)

The goal of any data distribution is to provide for increased availability, reliability, and improved query access time. On the other hand, as opposed to query access time,

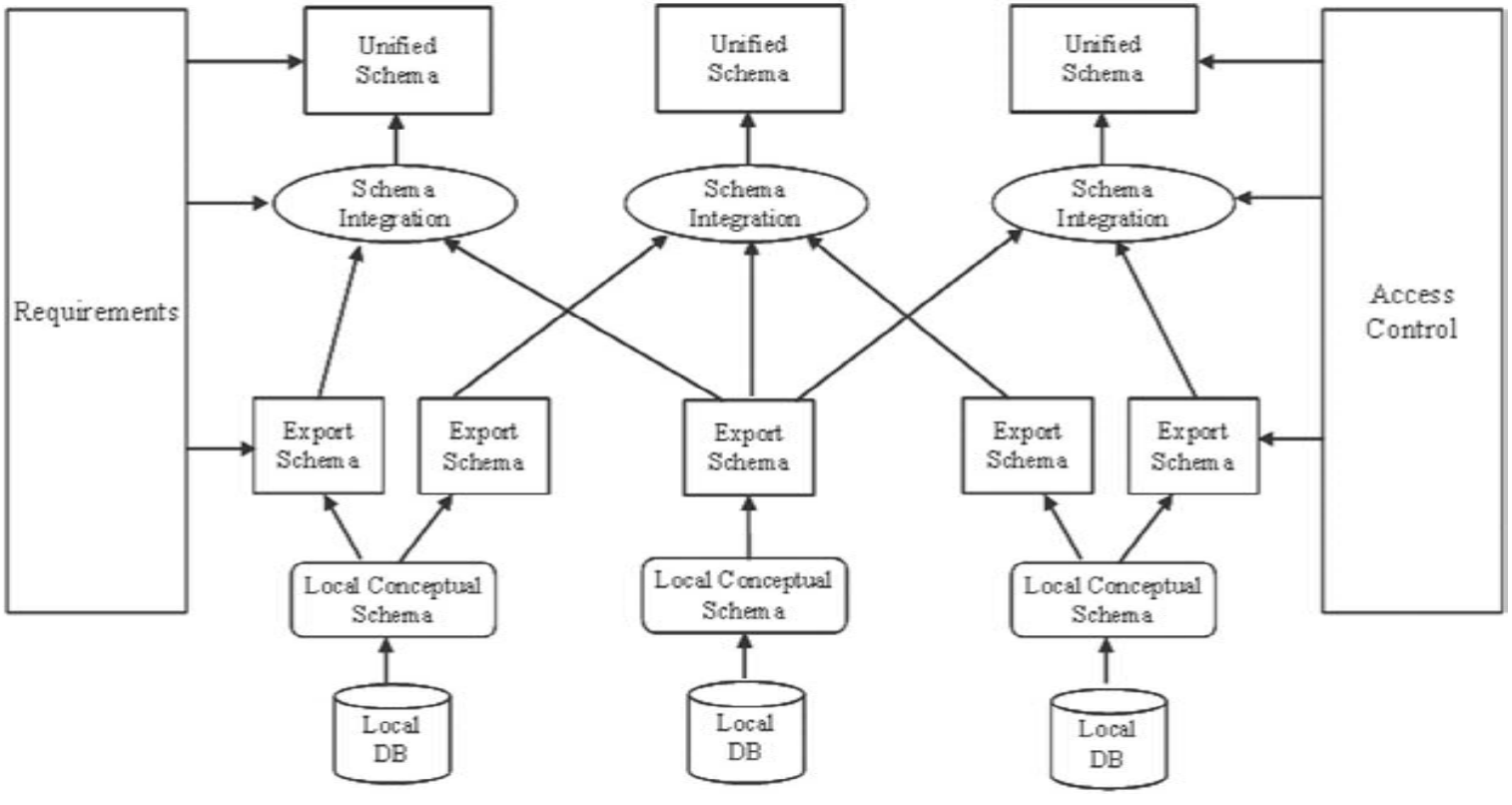


Figure 2.2 The bottom–up design process for a federated database system using schema integration.

distributed data generally takes more time for modification (update, delete, and insert). The impact of distribution on performance of queries and updates is well studied by researchers [Ceri87], [Özsu99]. It has been proved that for a given distribution design and a set of applications that query and update distributed data, determining the optimal data allocation strategy for database servers in a distributed system is an NP-complete problem [Eswaren74]. That is why most designers are not seeking the best data distribution and allocation design but one that minimizes some of the cost elements. We will examine the impact of distribution on the performance of queries and updates in more detail in Chapter 4.

2.1 DESIGN ALTERNATIVES

In this section, we will explore the details of each one of the design alternatives mentioned earlier.

2.1.1 Localized Data

This design alternative keeps all data logically belonging to a given DBMS at one site (usually the site where the controlling DBMS runs). This design alternative is sometimes called "not distributed."

2.1.2 Distributed Data

A database is said to be distributed if any of its tables are stored at different sites; one or more of its tables are replicated and their copies are stored at different sites; one or more of its tables are fragmented and the fragments are stored at different sites; and so on. In general, a database is distributed if not all of its data is localized at a single site.

2.1.2.1 ***Nonreplicated, Nonfragmented*** This design alternative allows a designer to place different tables of a given database at different sites. The idea is that data should be placed close to (or at the site) where it is needed the most. One benefit of such data placement is the reduction of the communication component of the processing cost. For example, assume a database has two tables called "EMP" and "DEPT." A designer of DDBMS may decide to place EMP at Site 1 and DEPT at Site 2. Although queries against the EMP table or the DEPT table are processed locally at Site 1 and Site 2, respectively, any queries against both EMP and DEPT together (join queries) will require a distributed query execution. The question that arises here is, "How would a designer decide on a specific data distribution?" The answer depends on the usage pattern for these two tables. This distribution allows for efficient access to each individual table from Sites 1 and 2. This is a good design if we assume there are a high number of queries needing access to the entire EMP table issued at Site 1 and a high number of queries needing access to the entire DEPT table issued at Site 2. Obviously, this design also assumes that the percentage of queries needing to join information across EMP and DEPT is low.

2.1.2.2 ***Fully Replicated*** This design alternative stores one copy of each database table at every site. Since every local system has a complete copy of the entire database,

all queries can be handled locally. This design alternative therefore provides for the best possible query performance. On the other hand, since all copies need to be in sync—show the same values—the update performance is impacted negatively. Designers of a DDBMS must evaluate the percentage of queries versus updates to make sure that deploying a fully replicated database has an overall acceptable performance for both queries and updates.

2.1.2.3 ***Fragmented or Partitioned*** Fragmentation design approach breaks a table up into two or more pieces called fragments or partitions and allows storage of these pieces in different sites.

There are three alternatives to fragmentation:

- Vertical fragmentation
- Horizontal fragmentation
- Hybrid fragmentation

This distribution alternative is based on the belief that not all the data within a table is required at a given site. In addition, fragmentation provides for increased parallelism, access, disaster recovery, and security/privacy. In this design alternative, there is only one copy of each fragment in the system (nonreplicated fragments). We will explain how each of the above fragmentation schemes works in Section 2.2.

2.1.2.4 ***Partially Replicated*** In this distribution alternative, the designer will make copies of some of the tables (or fragments) in the database and store these copies at different sites. This is based on the belief that the frequency of accessing database tables is not uniform. For example, perhaps Fragment 1 of the EMP table might be accessed more frequently than Fragment 2 of the table. To satisfy this requirement, the designer may decide to store only one copy of Fragment 2, but more than one copy of Fragment 1 in the system. Again, the number of Fragment 2 copies needed depends on how frequently these access queries run and where these access queries are generated.

2.1.2.5 ***Mixed Distribution*** In this design alternative, we fragment the database as desired, either horizontally or vertically, and then partially replicate some of the fragments.

2.2 FRAGMENTATION

As outlined earlier, fragmentation requires a table to be divided into a set of smaller tables called fragments. Fragmentation can be **horizontal, vertical**, or **hybrid** (a mix of horizontal and vertical). Horizontal fragmentation can further be classified into two classes: **primary horizontal fragmentation (PHF)** and **derived horizontal fragmentation (DHF)**. When thinking about fragmentation, designers need to decide on the degree of granularity for each fragment. In other words, how many of the table columns and/or rows should be in a fragment? The range of options is vast. At one end, we can have all the rows and all the columns of the table in one fragment. This obviously gives us a nonfragmented table; the grain is too coarse if we were planning to have at least one fragment. At the other end, we can put each data item (a single column value

for a single row) in a separate fragment. This grain obviously is too fine: it would be hard to manage and would add too much overhead to processing queries. The answer should be somewhere in between these two extremes. As we will explain later, the optimal solution depends on the type and frequency of queries that applications run against the table. In the rest of this section, we explore each fragmentation type and formalize the fragmentation process.

2.2.1 Vertical Fragmentation

Vertical fragmentation (VF) will group the columns of a table into fragments. VF must be done in such a way that the original table can be reconstructed from the fragments. This fragmentation requirement is called "reconstructiveness." This requirement is used to reconstruct the original table when needed. As a result, each VF fragment must contain the primary key column(s) of the table. Because each fragment contains a subset of the total set of columns in the table, VF can be used to enforce security and/or privacy of data. To create a vertical fragment from a table, a select statement is used in which "Column_list" is a list of columns from R that includes the primary key.

```
Select Column_list from R;
```

Example 2.1 Consider the EMP table shown in Figure 2.3. Let's assume that for security reasons the salary information for employees needs to be maintained in the company headquarters' server, which is located in Minneapolis.

To achieve this, the designer will fragment the table vertically into two fragments as follows:

```
Create table EMP_SAL as
     Select EmpID, Sal
     From EMP;
```

EmpID	Name	Loc	Sal	DOB	Dept
283948	Joe	LA	25,000	2/6/43	Maintenance
109288	Larry	New York	35,200	12/3/52	Payroll
284003	Moe	LA	43,000	7/12/56	Maintenance
320021	Sam	New York	53,500	8/30/47	Production
123456	Steve	Minneapolis	67,000	5/14/78	Management
334456	Jack	New York	55,000	5/30/67	Production
222222	Saeed	Minneapolis	34,000	4/27/59	Management

EMP Table

Figure 2.3 The nonfragmented version of the EMP table.

```
Create table EMP_NON_SAL as
      Select EmpID, Name, Loc, DOB, Dept
      From EMP;
```

EMP_SAL contains the salary information for all employees while EMP_NON_SAL contains the nonsensitive information. These statements generate the vertical fragments shown in Figure 2.4a, 2.4b from the EMP table.

After fragmentation, the EMP table will not be stored physically anywhere. But, to provide for fragmentation transparency—not requiring the users to know that the EMP table is fragmented—we have to be able to reconstruct the EMP table from its VF fragments. This will give the users the illusion that the EMP table is stored intact. To do this, we will use the following join statement anywhere the EMP table is required:

EmpID	Sal
283948	25,000
109288	35,200
284003	43,000
320021	53,500
123456	67,000
334456	55,000
222222	34,000

(a) EMP_Sal Fragment

EmpID	Name	Loc	DOB	Dept
283948	Joe	LA	2/6/43	Maintenance
109288	Larry	New York	12/3/52	Payroll
284003	Moe	LA	7/12/56	Maintenance
320021	Sam	New York	8/30/47	Production
123456	Steve	Minneapolis	5/14/78	Management
334456	Jack	New York	5/30/67	Production
222222	Saeed	Minneapolis	4/27/59	Management

(b) EMP_NON_Sal Fragment

Figure 2.4 The vertical fragments of the EMP table.

```
Select EMP_SAL.EmpID, Sal, Name, Loc, DOB, Dept
From EMP_SAL, EMP_NON_SAL
Where EMP_SAL.EmpID = EMP_NON_SAL.EmpID;
```

Note: This join statement can be used in defining a view called "EMP" and/or can be used as an in-line view in any select statement that uses the virtual (physically nonexisting) table "EMP."

2.2.2 Horizontal Fragmentation

Horizontal fragmentation (HF) can be applied to a base table or to a fragment of a table. Note that a fragment of a table is itself a table. Therefore, in the following discussion when we use the term table, we might refer to a base table or a fragment of the table. HF will group the rows of a table based on the values of one or more columns. Similar to vertical fragmentation, horizontal fragmentation must be done in such a way that the base table can be reconstructed (reconstructiveness). Because each fragment contains a subset of the rows in the table, HF can be used to enforce security and/or privacy of data. Every horizontal fragment must have all columns of the original base table. To create a horizontal fragment from a table, a select statement is used. For example, the following statement selects the row from R satisfying condition C:

```
Select * from R where C;
```

As mentioned earlier, there are two approaches to horizontal fragmentation. One is called primary horizontal fragmentation (PHF) and the other is called derived horizontal fragmentation (DHF).

2.2.2.1 ***Primary Horizontal Fragmentation*** Primary horizontal fragmentation (PHF) partitions a table horizontally based on the values of one or more columns of the table. Example 2.2 discusses the creation of three PHF fragments from the EMP table based on the values of the Loc column.

Example 2.2 Consider the EMP table shown in Figure 2.3. Suppose we have three branch offices, with each employee working at only one office. For ease of use, we decide that information for a given employee should be stored in the DBMS server at the branch office where that employee works. Therefore, the EMP table needs to be fragmented horizontally into three fragments based on the value of the Loc column as shown below:

```
Create table MPLS_EMPS as
     Select *
     From EMP
     Where Loc = 'Minneapolis';

Create table LA_EMPS as
     Select *
     From EMP
     Where Loc = 'LA';
```

```
Create table NY_EMPS as
     Select *
     From EMP
     Where Loc = 'New York';
```

This design generates three fragments, shown in Figure 2.5a,b,c. Each fragment can be stored in its corresponding city's server.

Again, after fragmentation, the EMP table will not be physically stored anywhere. To provide for horizontal fragmentation transparency, we have to be able to reconstruct the EMP table from its HF fragments. This will give the users the illusion that the EMP table is stored intact. To do this, we will use the following union statement anywhere the EMP table is required:

```
(Select * from MPLS_EMPS
Union
Select * from LA_EMPS)
     Union
     Select * from NY_EMPS;
```

EmpID	Name	Loc	Sal	DOB	Dept
123456	Steve	Minneapolis	67,000	5/14/78	Management
222222	Saeed	Minneapolis	34,000	4/27/59	Management

(a) MPLS_EMPS fragment

EmpID	Name	Loc	Sal	DOB	Dept
283948	Joe	LA	25,000	2/6/43	Maintenance
284003	Moe	LA	43,000	7/12/56	Maintenance

(b) LA_EMPS fragment

EmpID	Name	Loc	Sal	DOB	Dept
109288	Larry	New York	35,200	12/3/52	Payroll
320021	Sam	New York	53,500	8/30/47	Production
334456	Jack	New York	55,000	5/30/67	Production

(c) NY_EMPS fragment

Figure 2.5 The horizontal fragments of the EMP table with fragments based on Loc.

2.2.2.2 ***Derived Horizontal Fragmentation*** Instead of using PHF, a designer may decide to fragment a table according to the way that another table is fragmented. This type of fragmentation is called derived horizontal fragmentation (DHF). DHF is usually used for two tables that are naturally (and frequently) joined. Therefore, storing corresponding fragments from the two tables at the same site will speed up the join across the two tables. As a result, an implied requirement of this fragmentation design is the presence of a join column across the two tables.

Example 2.3 Figure 2.6a shows table "DEPT(Dno, Dname, Budget, Loc)," where Dno is the primary key of the table. Let's assume that DEPT is fragmented based on the department's city. Applying PHF to the DEPT table generates three horizontal fragments, one for each of the cities in the database, as depicted in Figure 2.6b,c,d.

Now, let's consider the table "PROJ," as depicted in Figure 2.7a. We can partition the PROJ table based on the values of Dno column in the DEPT table's fragments with the following SQL statements. These statements will produce the derived fragments from the PROJ table as shown in Figure 2.7b,c. Note that there are no rows in PROJ3, since department "D4" does not manage any project.

Dno	Dname	Budget	Loc
D1	Management	750,000	Minneapolis
D2	Payroll	500,000	New York
D3	Production	400,000	New York
D4	Maintenance	300,000	LA

(a) DEPT Table

Dno	Dname	Budget	Loc
D1	Management	750,000	Minneapolis

(b) MPLS_DEPTS Fragment

Dno	Dname	Budget	Loc
D2	Payroll	500,000	New York
D3	Production	400,000	New York

(c) NY_DEPTS Fragment

Dno	Dname	Budget	Loc
D4	Maintenance	300,000	LA

(d) LA_DEPTS Fragment

Figure 2.6 The fragments of the DEPT table with fragments based on Loc.

Pno	Pname	Budget	Dno
P1	Database Design	135,000	D2
P2	Maintenance	310,000	D3
P3	CAD/CAM	500,000	D2
P4	Architecture	300,000	D1
P5	Documentation	450,000	D1

(a) PROJ Table

Pno	Pname	Budget	Dno
P4	Architecture	300,000	D1
P5	Documentation	450,000	D1

(b) PROJ1

Pno	Pname	Budget	Dno
P1	Database Design	135,000	D2
P3	CAD/CAM	500,000	D2
P2	Maintenance	310,000	D3

(c) PROJ2

Figure 2.7 The PROJ table and its component DHF fragments.

```
Create table PROJ1 as
     Select Pno, Pname, Budget, PROJ.Dno
     From PROJ, MPLS_DEPTS
     Where PROJ.Dno = MPLS_DEPTS.Dno;

Create table PROJ2 as
     Select Pno, Pname, Budget, PROJ.Dno
     From PROJ, NY_DEPTS
     Where PROJ.Dno = NY_DEPTS.Dno;

Create table PROJ3 as
     Select Pno, Pname, Budget, PROJ.Dno
     From PROJ, LA_DEPTS
     Where PROJ.Dno = LA_DEPTS.Dno;
```

It should be rather obvious that all the rows in PROJ1 have corresponding rows in the MPLS_DEPTS fragment, and similarly, all the rows in PROJ2 have

corresponding rows in the NY_DEPTS fragment. Storing a derived fragment at the same database server where the deriving fragment is, will result in better performance since any join across the two tables' fragments will result in a 100% hit ratio (all rows in one fragment have matching rows in the other).

Example 2.4 For this example, assume that sometimes we want to find those projects that are managed by the departments that have a budget of less than or equal to 500,000 (department budget, not project budget) and at other times we want to find those projects that are managed by the departments that have a budget of more than 500,000. In order to achieve this, we fragment DEPT based on the budget of the department. All departments with a budget of less than or equal to 500,000 are stored in DEPT4 and other departments are stored in the DEPT5 fragment. Figures 2.8a and 2.8b show DEPT4 and DEPT5, respectively.

To easily answer the type of questions that we have outlined in this example, we should create two derived horizontal fragments of the PROJ table based on DEPT4 and DEPT5 as shown below.

```
Create table PROJ5 as
     Select Pno, Pname, Budget, PROJ.Dno
     From PROJ, DEPT4
     Where PROJ.Dno = DEPT4.Dno;

Create table PROJ6 as
     Select Pno, Pname, Budget, PROJ.Dno
     From PROJ, DEPT5
     Where PROJ.Dno = DEPT5.Dno;
```

Figure 2.9 shows the fragmentation of the PROJ table based on these SQL statements.

Dno	Dname	Budget	Loc
D3	Production	400,000	New York
D4	Maintenance	300,000	LA

(a) DEPT4

Dno	Dname	Budget	Loc
D1	Management	750,000	Minneapolis
D2	Payroll	500,000	New York

(b) DEPT5

Figure 2.8 The DEPT table fragmented based on Budget column values.

Pno	Pname	Budget	Dno
P2	Maintenance	310,000	D3

(a) PROJ5

Pno	Pname	Budget	Dno
P1	Database Design	135,000	D2
P3	CAD/CAM	500,000	D2
P4	Architecture	300,000	D1
P5	Documentation	450,000	D1

(b) PROJ6

Figure 2.9 The derived fragmentation of PROJ table based on the fragmented DEPT table.

2.2.3 Hybrid Fragmentation

Hybrid fragmentation (HyF) uses a combination of horizontal and vertical fragmentation to generate the fragments we need. There are two approaches to doing this. In the first approach, we generate a set of horizontal fragments and then vertically fragment one of more of these horizontal fragments. In the second approach, we generate a set of vertical fragments and then horizontally fragment one or more of these vertical fragments. Either way, the final fragments produced are the same. This fragmentation approach provides for the most flexibility for the designers but at the same time it is the most expensive approach with respect to reconstruction of the original table.

Example 2.5 Let's assume that employee salary information needs to be maintained in a separate fragment from the nonsalary information as discussed above. A vertical fragmentation plan will generate the EMP_SAL and EMP_NON_SAL vertical fragments as explained in Example 2.1. The nonsalary information needs to be fragmented into horizontal fragments, where each fragment contains only the rows that match the city where the employees work. We can achieve this by applying horizontal fragmentation to the EMP_NON_SAL fragment of the EMP table. The following three SQL statements show how this is achieved.

```
Create table NON_SAL_MPLS_EMPS as
     Select *
     From EMP_NON_SAL
     Where Loc = 'Minneapolis';

Create table NON_SAL_LA_EMPS as
     Select *
     From EMP_NON_SAL
     Where Loc = 'LA';
```

```
Create table NON_SAL_NY_EMPS as
     Select *
     From EMP_NON_SAL
     Where Loc = 'New York';
```

The final distributed database is depicted in Figure 2.10.

Observation: The temporary EMP_NON_SAL fragment is not physically stored anywhere in the system after it has been horizontally fragmented. As a result, one can bypass generating this fragment by using the following set of SQL statements to generate the required fragments directly from the EMP table.

EmpID	Sal
283948	25,000
109288	35,200
284003	43,000
320021	53,500
123456	67,000
334456	55,000
222222	34,000

EMP_Sal

EmpID	Name	Loc	DOB	Dept
123456	Steve	Minneapolis	5/14/78	Management
222222	Saeed	Minneapolis	4/27/59	Management

NON_Sal_MPLS_EMPS

EmpID	Name	Loc	DOB	Dept
283948	Joe	LA	2/6/43	Maintenance
284003	Moe	LA	7/12/56	Maintenance

NON_Sal_LA_EMPS

EmpID	Name	Loc	DOB	Dept
109288	Larry	New York	12/3/52	Payroll
320021	Sam	New York	8/30/47	Production
334456	Jack	New York	5/30/67	Production

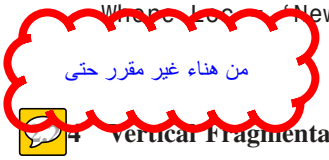
NON_Sal_NY_EMPS

Figure 2.10 The fragments of the EMP table.

```
Create table NON_SAL_MPLS_EMPS as
     Select EmpID, Name, Loc, DOB, Dept
     From EMP
     Where Loc = 'Minneapolis';

Create table NON_SAL_LA_EMPS as
     Select EmpID, Name, Loc, DOB, Dept
     From EMP
     Where Loc = 'LA';

Create table NON_SAL_NY_EMPS as
     Select EmpID, Name, Loc, DOB, Dept
     From EMP
     Where Loc = 'New York';
```



4 Vertical Fragmentation Generation Guidelines

There are two approaches to vertical fragmentation design—grouping and splitting—proposed in the literature [Hoffer75] [Hammer79] [Sacca85]. In the remainder of this section, we will first provide an overview of these two options and then present more detail for the splitting option.

2.2.4.1 Grouping Grouping is an approach that starts by creating as many vertical fragments as possible and then incrementally reducing the number of fragments by merging the fragments together. Initially, we create one fragment per nonkey column, placing the nonkey column and the primary key of the table into each vertical fragment. This first step creates as many vertical fragments as the number of nonkey columns in the table. Most of the time, this degree of fragmentation is too fine and impractical. The grouping approach uses joins across the primary key, to group some of these fragments together, and we continue this process until the desired design is achieved. Aside from needing to fulfill the requirements for one or more application, there are very few restrictions placed upon the groups (fragments) we create in this approach. For example, the same nonkey column can participate in more than one group—that is, groups can have overlapping (nonkey) columns. If this "overlap" does occur, obviously, it will add to the overhead of replication control in a distributed DBMS system. As a result, grouping is not usually considered a valid approach for vertical fragmentation design. For more details on grouping see [Hammer79] and [Sacca85]. Hammer and Niamir introduced grouping for centralized DBMSs and Sacca and Wiederhold discussed grouping for distributed DBMSs.

2.2.4.2 Splitting Splitting is essentially the opposite of grouping. In this approach, a table is fragmented by placing each nonkey column in one (and only one) fragment, focusing on identifying a set of required columns for each vertical fragment. As such, there is no overlap of nonprimary key columns in the vertical fragments that are created using splitting. Hoffer and Severance [Hoffer75] first introduced splitting for centralized systems, while Navathe and colleagues [Navathe84] introduced splitting for

2.2.8 Replication

During the database design process, the designer may decide to copy some of the fragments or tables to provide better accessibility and reliability. It should be obvious that the more copies of a table/fragment one creates, the easier it is to query that table/fragment. On the other hand, the more copies that exist, the more complicated (and time consuming) it is to update all the copies. That is why a designer has to know the frequency by which a table/fragment is queried versus the frequency by which it is modified—via inserts, updates, or deletes. As a rule of thumb, if it is queried more frequently than it is modified, then replication is advisable. Once we store more than one copy of a table/fragment in the distributed database system, we increase the probability of having a copy locally available to query.

Having more than one copy of a fragment in the system increases the resiliency of the system as well. That is because the probability of all copies failing at the same time is very low. In other words, we can still access one of the copies even if some of the copies have failed: that is, of course, if all copies of a fragment show the same values. Therefore, this benefit comes with the additional cost of keeping all copies identical. This cost, which could potentially be high, consists of total storage cost, cost of local processing, and communication cost. Note that the copies need to be identical only when the copies are online (in service). We will discuss the details of how copies are kept in sync as part of the replication control (in Chapter 7). We will also discuss calculating total cost of queries/updates as part of managing transactions (in Chapter 3) and query optimization (in Chapter 4).

2.3 DISTRIBUTION TRANSPARENCY

Although a DDBMS designer may fragment and replicate the fragments or the tables of a system, the users of such a system should not be aware of these details. This is what is known as **distribution transparency**. Distribution transparency is one of the sought after features of a distributed DBE. It is this transparency that makes the system easy to use by hiding the details of distribution from the users. There are three aspects of distribution transparency—location, fragmentation, and replication transparencies.

2.3.1 Location Transparency

The fact that a table (or a fragment of table) is stored at a remote site in a distributed system should be hidden from the user. When a table or fragment is stored remotely, the user should not need to know which site it is located at, or even be aware that it is not located locally. This provides for location transparency, which enables the user to query any table (or any fragment) as if it were stored locally.

2.3.2 Fragmentation Transparency

The fact that a table is fragmented should be hidden from the user. This provides for fragmentation transparency, which enables the user to query any table as if it were intact and physically stored. This is somewhat analogous to the way that users of a SQL view are often unaware that they are not using an actual table (many views are actually defined as several union and join operations working across several different tables).

2.3.3 Replication Transparency

The fact that there might be more than one copy of a table stored in the system should be hidden from the user. This provides for replication transparency, which enables the user to query any table as if there were only one copy of it.

2.3.4 Location, Fragmentation, and Replication Transparencies

The fact that a DDBE designer may fragment a table, make copies of the fragments, and store these copies at remote sites should be hidden from the user. This provides for complete distribution transparency, which enables the user to query the table as if it were physically stored at the local site without being fragmented or replicated.

2.4 IMPACT OF DISTRIBUTION ON USER QUERIES

Developers of a distributed DBMS try to provide for location, fragmentation, and replication transparencies to their users. This is an attempt to make the system easier to use. It is obvious that in order to provide for these transparencies, a DDBMS must store distribution information in its global data dictionary and use this information in processing the users' requests. It is expensive to give users complete distribution transparency. In such a system, although the users query the tables as if they were stored locally, in reality their queries must be processed by one or more database servers across the network. Coordinating the work of these servers is time consuming and hard to do. In Chapter 1, we discussed the issues related to distributed query execution. We will also address the performance impact of providing complete distribution transparency in Chapter 4. In the rest of this section, we will outline the impact of distribution on a user's queries.

Example 2.9 Let's assume that our database contains an employee table as defined below:

```
EMP (Eno, Ename, Sal, Tax, Mgr, Dno)
```

The column "Eno" is the primary key of this table. The column "Dno" is a foreign key that tracks the department number of the department in which an employee works. Suppose we have horizontally fragmented EMP into EMP1 and EMP2, where EMP1 contains only employees who work in a department with a department number less than or equal to 10, while EMP2 contains only employees who work in the departments with a department number greater than 10. Assume EMP2 has been replicated and there are two copies of it. Also assume that the company owns three database servers—one server is in Minneapolis (Site 1), one server is in St. Paul (Site 2), and the third server is located in St. Cloud (Site 3). The company decides to store EMP1 at Site 1, one copy of EMP2 at Site 2, and the other copy of EMP2 at Site 3. To see the impact of this database design on a user's queries, let's discuss a very simple query that finds salary information for "Jones," who is an employee with employee number 100. If the system were a centralized DBMS, a user would run the following SQL statement to find Jones' information:

```
Select * from EMP where Eno = 100;
```

In this example, our system is a distributed system and the table has been fragmented and replicated. How would a user write the query in this distributed system? The answer depends on whether or not the system provides for location, fragmentation, and replication transparencies. As mentioned in Chapter 1, the global data dictionary (GDD) stores distribution information for the environment. The impact of data distribution on user queries depends on how much of the distribution information is stored in the GDD and how many types of transparency are provided to the user.

Let's consider three different scenarios, with different degrees of transparency being provided to the user. For the first case, think of a GDD implementation that does not store any distribution information—the GDD is basically nonexistent. In this case, the design, implementation, and administration of the GDD would be trivial, but the GDD would provide absolutely no transparency to the user. This is clearly the simplest GDD implementation possible, but from the user's point of view, it is the hardest to use. The harder the system is to use, the less likely it will be used in the real world! Suppose we had the other extreme situation instead. This other case would be a very powerful GDD providing location, replication, and fragmentation transparencies to the user. While this would be nice for the user and it is the easiest for the user to use, this other extreme would obviously require a more complicated design and implementation, and would also be less trivial to administer. The more complicated a system is to implement and administer, the less likely it is to become ubiquitous. Somewhere between these two extremes, we are likely to find the right balance of transparency provided versus complexity required. In Sections 2.4.1, 2.4.2, and 2.4.3, we will attempt to illustrate the types of trade-offs that need to be considered as we move from case to case between these extremes.

2.4.1 No GDD—No Transparency

If the GDD does not contain any information about data distribution, then the users will need to be aware of where data is located, how it is fragmented and replicated, and where these fragments and replicas are stored. This means that the users need to know what information is stored in each of the local systems and then they need to incorporate this information into their queries. Figure 2.30 depicts the contents of the local data dictionaries at each of the three sites and the GDD.

Since the GDD does not store any distribution information, the users need to know this information. Figure 2.31 shows the program that a user writes to retrieve salary for Jones. **Note:** We have used the notation "EMP1@Site1" to indicate the need to run the SQL command against the EMP1 fragment at Site 1. Obviously, this is not a valid SQL statement and will **not** be parsed correctly by any commercial DBMS. We are assuming, however, that our DDBMS has a parser that understands this notation, translates it into the correct syntax, and actually sends the correct SQL statements to the right database servers. The notation /*. . . */ represents a comment in SQL.

In this program, the user assumes Jones is in EMP1 and queries EMP1 looking for Jones. The user had to "hard-code" the location for the table indicating which site contained the EMP1 table. Since there is only one copy of EMP1 in the system, the user specified Site 1. If the employee is not found there, then the employee might be in EMP2. We have two copies of EMP2 (at Site 2 and Site 3). There is no need to look in both of them since these are copies of the same fragment. The user has to decide which site to query for this fragment. In our example, the user has chosen Site 3. For this simple query, the user needed to write a rather long program,

```
insert into Emp3 values(100, $name, 15) @site 7;
insert into Emp4 values(100, $sal, $tax, $mgr) @site 4;
insert into Emp4 values (100, $sal, $tax, $mgr) @site 8;
```

This program resembles the program we presented in Section 2.5.2. However, since there are two copies of each fragment, we need to explicitly insert Smith's information into each copy of the new fragment and also explicitly delete Smith's information from each copy of the old fragment. Once again, we should really explicitly add transactions or locks to this program to ensure the integrity of the system. Notes 1 and 2 from Section 2.5.2 apply here as well.

2.6 SUMMARY

Distribution design in a distributed database management system is a top–down process. Designers start with the data access requirements of all users of the distributed system and design the contents of the distributed database as well as the global data dictionary to satisfy these requirements. Once the GDD contents are determined, the designers decide how to fragment, replicate, and allocate data to individual database servers across the network. In this chapter, we outlined the necessary steps for such a design.

We outlined the steps for creating horizontal fragments and vertical fragments of a table based on a set of requirements. We examined the rules for correct fragmentation design (such as completeness, reconstructiveness, and disjointness). We briefly discussed the impact of distribution on complexity of user programs to access and update distributed information.

2.7 GLOSSARY

Access Frequency A measurement reflecting how often an application accesses the columns of a table during a defined period of time, for example, "fifteen accesses per day."

Affinity A measure of closeness between the columns of a table as it relates to a given application or a set of applications.

Affinity Matrix A matrix that contains the affinity of all columns of a table for all applications that refer to them.

Bond Energy Algorithm (BEA) The algorithm that calculates a metric that indicates the benefits of having two or more columns to be put in the same vertical partition.

Bottom–Up A methodology that creates a complex system by integrating system components.

Clustered Affinity Matrix A matrix that is used for vertical partitioning of columns of a table based on the Bond Energy Algorithm.

Derived Horizontal Fragmentation A horizontal fragmentation approach that fragments rows of a table based on the values of columns of another table.

Distribution Transparency A type of transparency that hides the distribution details of database tables or fragments from the end users.

Fragmentation Transparency A type of transparency that hides fragmentation details of database tables or fragments from the end users.

Fragment A partition of a table.

Global Conceptual Model A repository of information such as location, fragmentation, replication, and distribution for a distributed database system.

Global Data Dictionary (GDD) The portion of global conceptual schema that contains dictionary information such as table name, column names, view names, and so on.

Horizontal Fragmentation The act of partitioning (grouping) rows of a table into a smaller set of tables.

Hybrid Fragmentation The application of horizontal fragmentation to vertical fragments of a table, or the application of vertical fragmentation to horizontal fragments of a table.

Local Conceptual Model The conceptual schema that users of a local DBMS employ.

Location Transparency A transparency that hides location of database tables or fragments from the end users.

Minterm Predicate A predicate that contains two or more simple predicates.

Partitions Either vertical or horizontal fragments of a table.

Primary Horizontal Fragmentation A horizontal fragmentation of a table based on the value of one of its columns.

Replication Transparency A transparency that hides the fact that there might be more than one copy of a database table or fragments from the end users.

Schema Integration The act of integrating local conceptual schemas of the component database systems into a cohesive global conceptual schema.

Simple Predicate A condition that compares a column of a table against a given value.

Top–Down A database design methodology that starts with the requirements of a distributed system (the global conceptual schema) and creates its local conceptual schemas.

Unified Schema An integrated, nonredundant, and consistent schema for a set of local database systems.

Usage Matrix A matrix that indicates the frequency of usage of columns of a table by a set of applications.

Vertical Fragmentation The act of partitioning the columns of a table into a set of smaller tables.

REFERENCES

[Bobak96] Bobak, A., *Distributed and Multi-Database Systems*, Artech House, Boston, MA, 1996.

[Ceri84] Ceri, S., and Pelagatti, G., *Distributed Databases—Principles and Systems*, McGraw-Hill, New York, 1984.

[Ceri87] Ceri, S., Pernici, B., and Wiederhold, G., "Distributed Database Design Methodologies," *Proceedings of IEEE*, Vol. 75, No. 5, pp. 533–546, May 1987.

[Eswaren74] Eswaren, K., "Placement of Records in a File and File Allocation in a Computer Network," in *Proceedings of Information Processing Conference*, Stockholm, pp. 304–307, 1974.

[Hammer79] Hammer, M., and Niamir, B., "A Heuristic Approach to Attribute Partitioning," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 93–101, Boston, May 1979.

[Hoffer75] Hoffer, H., and Severance, D., "The Use of Cluster Analysis in Physical Database Design," in *Proceedings of the First International Conference on Very Large Databases*, Framingham, MA, pp. 69–86, September 1975.

[McCormick72] McCormick, W., Schweitzer, P., and White T., "Problem Decomposition and Data Reorganization by a Clustering Technique," *Operations Research*, Vol. 20, No. 5, pp. 993–1009, 1972.

[Navathe84] Navathe, S., Ceri, S., Wiederhold, G., and Dou, J., "Vertical Partitioning Algorithms for Database Design," *ACM Transactions on Database Systems*, Vol. 9, No. 4, pp. 680–710, December 1984.

[Özsu99] Özsu, M., and Valduriez, P., *Principles of Distributed Database Systems*, Prentice Hall, Englewood Cliffs, NJ, 1999.

[Sacca85] Sacca, D., and Wiederhold, G., "Database Partitioning in a Cluster of Processors," *ACM Transactions on Database Systems*, Vol. 10, No. 1, pp. 29–56, October 1985.

EXERCISES

Provide short (but complete) answers to the following questions.

2.1 Assume EMP has been fragmented as indicated in Figure 2.36. Also, assume the system **does not** provide for any transparencies. Write a SQL-like program that deletes the employee indicated by $eno from the database. Make sure you perform all the necessary error checking.

2.2 Answer true or false for the statements in Figure 2.37.

2.3 For the EMP table fragmented in Example 2.5, write a **single** SQL statement that reconstructs the original EMP table from its fragments.

2.4 An Employee table has the following relational scheme: "Employee (name, sal, loc, mgr)," where name is the primary key. The table has been horizontally fragmented into SP and MPLS fragments. SP has employees who work in St. Paul and MPLS contains all employees who work in Minneapolis. Each fragment is stored in the city where the employees are located. Assume transactions only enter the system in NY and there are no employees in NY. Write down the local schemas and global schema and indicate in which cities these schemas are located for the following three cases:

(A) The system does not provide for any transparencies.

(B) The system provides for location and replication transparencies.

(C) The system provides for location, replication, and fragmentation transparencies.

2.5 Consider table "EMP(EmpID, Name, Sal, Loc, Dept)." There are four applications running against this table as shown below. Design an optimal vertical

#	Statement	True/ False
1	In SQL, vertical fragments are created using the project statement.	
2	The external schemas define the database as the end users see it.	
3	Tuple is another word for a row in a relational database.	
4	Federated databases have three levels of schemas.	
5	Federated databases are formed from the Bottom–Up.	
6	Referential integrity enforced by a DDBMS does not span sites.	
7	Horizontal fragments need to be disjoint.	
8	Distributed DBMSs are formed Top–Down.	
9	Global data dictionary may be copied at some sites in a distributed DBMS.	
10	There are as many physical fragments as there are minterm predicates of a minimal and complete distribution design.	

Figure 2.37 True or false questions.

fragmentation strategy that satisfies the needs of these applications. Show steps for arriving at the answer.

A1: "Select EmpID, Sal From EMP;"
A2: "Select EmpID, Name, Loc, Dept From EMP Where Dept = 'Eng';"
A3: "Select EmpID, Name, Loc, Dept From EMP Where Loc = 'STP';"
A4: "Select EmpID, Name, Loc, Dept from EMP Where Loc = 'MPLS';"

3

DATABASE CONTROL

Database control is one of most challenging tasks that every **database administrator (DBA)** faces. Controlling a database means being able to provide correct data to valid users and applications. A data item must satisfy the correctness condition(s) that have been defined for it. The correctness conditions that are attached to a piece of data are called **constraints, semantic integrity rules, integrity constraints**, or **assertions**. We will use these terms interchangeably.

Let's consider the table "EMP (ENO, Ename, Sal, DNO)," where we store information about the employees of an organization. In this table, DNO represents the department in which the employee works.

Associated with DNO, there are a number of correctness criteria such as:

- DNO is an integer.
- DNO must be between 100 and 999.
- DNO must have a value for each employee.
- DNO value for a given employee must match the value of DNO in the DEPT table.

If any of these assertions for a given employee is violated, then the employee is not a valid employee. Semantic integrity rules must be defined before they can be enforced. Enforcing these assertions is the responsibility of the semantic integrity control module of the local DBE (see Chapter 1). These topics are discussed here under the topic of **semantic integrity control** or simply integrity control.

In addition to definition and enforcement of semantic integrity rules, the access to the database must be controlled. Only authenticated users with the proper authorization can be allowed to access the contents of the database. A DBA creates a list of valid

Distributed Database Management Systems by Saeed K. Rahimi and Frank S. Haug

users and defines the scope of their rights using the control language of the DBE. These issues are discussed under the topic of **access control**. Imposters and hackers must not be able to access and use the information in the database. An environment must have a sophisticated security mechanism in place to be able to thwart today's intruders. **Security** in a centralized DBE, and more importantly in a DDBE, is very difficult and costly to implement. We will discuss security issues and different threat types in Chapter 9.

There are two aspects to access control. The first has to do with **authentication** and the second with **access rights**. Authentication is used to make sure only preapproved applications and users can work with the database. Most of today's systems have a multilevel authentication system that checks the users' access to the database contents. Once a valid user has connected to the right database, the privileges that the user has must also be controlled. This is typically enforced by the DBE. Access rights of a given user can be defined, controlled, and enforced by the system to ensure that the user can only work with an allowed portion of the database and perform only allowed functions.

3.1 AUTHENTICATION

Authentication in a DBE guarantees that only legitimate users have access to data resources in a DBE. At the highest level of authentication, access to the client computer (the client is the front-end to the database server) or the database server is controlled. This is typically achieved by a combination of user-name/password and is enforced by the operating system. Other and more sophisticated approaches such as biometrics can also be used (see Chapter 9 for more detail).

Once the user is connected to the computer, accessing the DBE software is controlled by the second level of authentication. At this level, the user may use credentials that have been assigned by the DBE or the DBE can trust the operating system authentication and allow connection to the DBE software. The first approach is known as **database authentication** while the second is known as **operating system authentication**. SQL Server and Oracle have similar concepts, although they are not exactly the same concepts. A **login** (or **account**) is used to control access to the DBMS server while a **user** is used to control access to a database managed by a server. Sometimes in a DBE these concepts are implemented as separate things, sometimes a single implementation is used to represent both concepts.

In Oracle 10 g, a user account can use operating system (or external) authentication for validation. This allows operating system users to connect to the databases controlled by Oracle server. Oracle trusts the operating system's authentication and allows the current user to connect to a given database. Oracle can also use a database user-name and a password (maintained in the database) to authenticate an account. Assuming "saeed" is a database user-name with password "secret," the user will connect to a given database by providing the combination of saeed/secret and the database name.

In SQL Server 2005, the connection to the DBMS software is controlled by what is known as a login. A login is an authenticated connection to the server. An operating system user may login to a database server using two approaches similar to the way Oracle authenticates users. A login could be authenticated by the operating system (Windows) or by SQL Server. Again, operating system authentication is based on

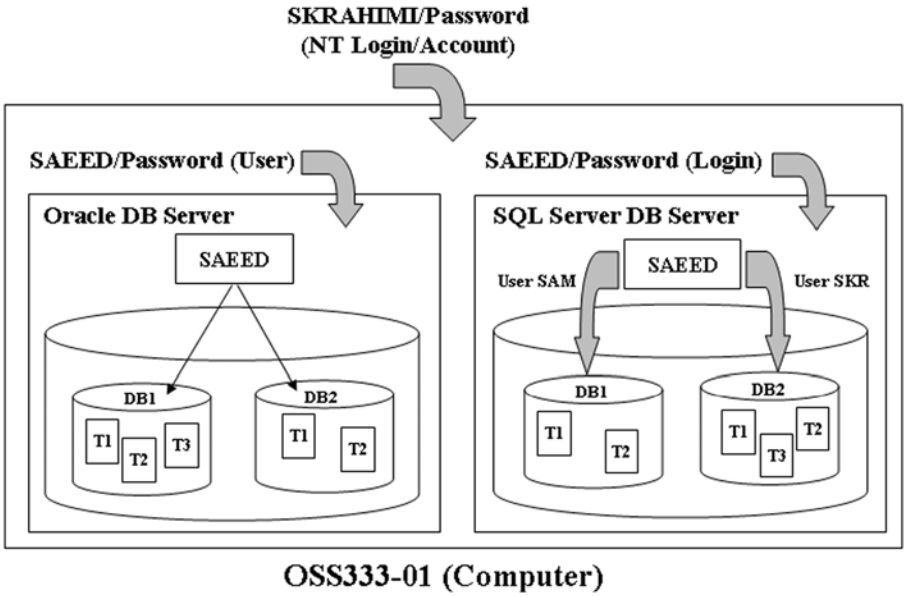


Figure 3.1 Multilevel security example.

the trust relationship between the operating system and DBMS software. SQL Server authentication requires a login-name and a password. Regardless of how a login has been established, to work with a given database, a login must use a database user. A database user represents a login inside a database. As a result, a given login can work with multiple databases using multiple database users. Figure 3.1 depicts multilevel authentication approaches for Oracle and SQL Server. In a DBE, the DBA is responsible for managing user accounts, logins, and user-names.

3.2 ACCESS RIGHTS

In relational database systems, what a database user can do inside the database is controlled by the access rights that are given to that user. A user's access rights specify the privileges that the user has. For example, user "saeed" may have the right to create a table, to drop a table, or to add rows to a table. SQL, as the standard for relational model databases, can be used to specify at a very fine level of granularity (column level) the operations allowed and disallowed for a given user.

In any large database environment, there are many users, many databases, many tables with many columns, and many other database objects. As a result, in large systems, assignment of individual privileges to individual users is a time-consuming and error-prone task. To reduce the overhead associated with management of rights for a large system, the concept of a **role** is used. A role is a construct that can be given certain authorities within a database (or the server). Figure 3.2 shows the creation of a role called manager. This role is then given some privileges and, finally, user1 is given the manager role. After executing these statements, user1 will have all privileges that have been assigned to the role 'manager'—user1 plays the role of a manager.

```
Create role manager;
Commit;
Grant select, insert, delete on employee to manager;
Grant insert, update on project to manager;
Grant select on department to manager;
Grant execute on payroll to manager;
Commit;
Grant manager to user1;
Commit;
```

Figure 3.2 Creation of a role and its assignment to the user "user1."

3.3 SEMANTIC INTEGRITY CONTROL

A DBMS must have the ability to specify and enforce correctness assertions in terms a set of semantic integrity rules. The semantic integrity service (Semi-S), which we discussed in Chapter 1, Section 1.3.1, is used to define and enforce the semantic integrity rules for the system. This is true for both relational and nonrelational database systems, such as network, hierarchical, and object-oriented systems. For instance, the need to enforce uniqueness of the primary key (or its equivalent counterpart) is the same in all database types, although the approaches used by different systems are different. Similarly, the need to enforce the existence of a primary key value for a given foreign key is the same for all database types and data models.

Figure 3.3a depicts a network database DDL statement that declares a record type called "PART" with two fields, "PART-NO" and "AMOUNT." PART-NO is considered the primary key for the record since the field is not allowed to have duplicate values. PART-NO has an alphanumeric data type of length 5 and AMOUNT has a numeric data type that has four significant digits and two decimal places. The same record definition is shown in Figure 3.3b for a relational system.

Figure 3.4a shows an example of referential integrity across two record types in a network database type. For this system, the SET definition statement guarantees (forces the Semi-S to check for) a valid association between the two record types, PART and SUPPLIER. The same requirement in a relational system is shown in Figure 3.4b.

```
NETWORK DDL
RECORD NAME IS PART
              DUPLICATES ARE NOT ALLOWED FOR PART-NO
              PART-NO       PICTURE X99999
              AMOUNT        PICTURE 9999V99
                                  (a)

RELATIONAL DDL
CREATE TABLE PART( PART-NO CHAR(5) NOT NULL PRIMARY KEY,
                   AMOUNT DECIMAL(6,2));
                                  (b)
```

Figure 3.3 Comparing network and relational model DDL statements.

```
Network Model
RECORD NAME IS PART
              DUPLICATES ARE NOT ALLOWED FOR PART-NO
              PART-NO        TYPE IS FIXED 6
              PART-NAME      TYPE IS CHAR 20
              QTY            TYPE IS FIXED 5

RECORD NAME IS SUPPLIER
              DUPLICATES ARE NOT ALLOWED FOR SUPPLIER-NO
              SUPPLIER-NO    TYPE IS FIXED 4
              SUPPLIER-CITY  TYPE IS CHAR 30
              STATUS         TYPE IS FIXED 4

SET NAME IS PART-SUPPLIER
              OWNER          IS SUPPLIER
              MEMBER         IS PART
              INSERTION      IS AUTOMATIC (MANUAL)
              RETENTION      IS OPTIONAL (MANDATORY, FIXED)
                                 (a)

Relational Model
CREATE TABLE PART (PART-NO DECIMAL (6) NOT NULL PRIMARY KEY,
              PART-NAME      CHAR(20) NOT NULL,
              QTY            DECIMAL(5)    NOT NULL,
              SNO            DECIMAL(4));

CREATE TABLE SUPPLIER (SNO DECIMAL (4) NOT NULL PRIMARY KEY,
              SCITY          CHAR(30) NOT NULL,
              STATUS         DECIMAL(4)    NOT NULL);

ALTER TABLE PART
ADD CONSTRAINT "PART-SUPPLIER-FKEY" FOREIGN KEY(SNO)
REFERENCES SUPPLIER(SNO);
                                 (b)
```

Figure 3.4 Specifying referential integrity in network and relational models.

For the relational system, the referential integrity requirement is added to the table definitions using the "ALTER TABLE" statement after the two tables have been created.

When one or more semantic integrity rules are violated, Semi-S can report, reject, or try to correct the query or transaction that is performing an illegal operation. For example, Semi-S can evaluate the SQL query in Figure 3.5a and reject the query since it knows the Sal column has a numeric data type and the value being compared to it is a character string. In a smarter DBE, the Semi-S subsystem may try to correct the query by removing the single quotes from around 40000 and running the query. This issue has been resolved for the SQL statement in Figure 3.5b, but there is still a potential issue with the T1 table's existence (or lack thereof). If T1 exists, the statement is fine; otherwise, an "object not found" error message must be issued by the Semi-S.

```
Select Last_Name, Sal
From Emp
Where Sal >= '40000';
```
(a)

```
Select Last_Name, Sal
From T1
Where Sal >= 40000;
```
(b)

Figure 3.5 Example of two semantically incorrect SQL statements.

3.3.1 Semantic Integrity Constraints

A relational DBE supports four basic types of semantic integrity constraints or rules:

- Data type constraints
- Relation constraints
- Referential constraints
- Explicit constraints

The first three types are called **relational constraints** and are inherited from the entity relationship model (ERM): see Chapter 10. These integrity rules are an integral part of the relational model; see Chapter 11. These rules are often automatically generated by our data modeling tools—however, the last rule needs to be specified by the user explicitly in the relational model.

3.3.1.1 Relational Constraints **Data type constraints** are semantic integrity rules that specify the **data type** for columns of relational tables. These are inherited from the ERM since the domains are tied to the underlying domains on which the attributes of an entity are defined in the ERM. Some relational database systems may allow specification of user-defined data types. A data type constrains the range of values and the type of operations that we can apply to the column to which the data type is attached. A **user-defined data type** is a data type that is based on a system data type and creates a specific data type on top of it. Examples of system-defined data types supported by most DBMS are Date, Integer, Decimal (X,Y), Float, Char(X), Varchar(X), Enumerated data types such (‘M’, ‘F’) or (‘Sat’, ‘Sun’, ‘Mon’, ‘Tue’, ‘Wed’, ‘Thu’, ‘Fri’), and Range data types such as (1–31). Figures 3.6a and 3.6b show two different database implementations of the same user-defined data type concept, for SQL Server 2005 and Oracle 11 g, respectively. In this example, we create the “salary” data type, basing it on the “numeric” system data type for SQL Server and the “number” data type for Oracle.

Relation constraints are the methods used to define a relation (or a table). The following codifies a table called Employee, which keeps track of the employees of an organization.

```
SQL Server User Defined Data Type:
CREATE TYPE salary FROM NUMERIC (8,2) NOT NULL
Go

CREATE TABLE EMP(Name       varchar (20)  NOT NULL ,
                 Comp       salary        NOT NULL,
                 Age        smallint      NULL )
Go
                          (a)

Oracle 11g User Defined Data Type:
CREATE TYPE salary AS OBJECT (salary number(8,2));

CREATE TABLE EMP (Name      varchar(20) NOT NULL,
                  Comp      salary      NOT NULL,
                  Age       number(3,0))NULL;
                          (b)
```

Figure 3.6 Example of a user-defined data type definition in SQL Server and Oracle.

```
Create table Employee (
     Emp#          integer not null,
     Name          varchar(20) not null,
     Emp_Nickname  char(10),
     DOB           date check (DOB LIKE 'MM-DD-YYYY'),
     Sex           char(1) in ('M', 'F'),
     Primary key (Emp#),
     Unique Emp_Nickname);
```

This code snippet conveys the following constraints:

- The table name "Employee" is unique within a database. This is called a relation constraint and is inherited from the fact that in the ERM there is a unique entity type called "Employee."
- Employees as objects of the real world are kept track of in the database as a set of rows in the Employee table.
- The Employee table has five columns: Emp#, Name, Emp_Nickname, DOB, and Sex. Each column has a data type constraint, may have a null or not null constraint, may have an enumeration constraint, and may have a range constraint.
- The Emp# column must be an integer and must have a value.
- The Name column is of type varchar(20) and must have a value. Since "varchar(20)" is used as the data type, the Name cannot be longer than 20 characters. If the Name is shorter than 20 characters, then the actual length of the name will be used.
- The DOB column is of type Date. This column may be left as null. If the column has a value for date it has to follow the format 'MM-DD-YYYY'. Otherwise, the row is not validated.

- The Sex column is a one-character field and can be left as null. The Sex column has an enumeration constraint on it. This constraint limits the value of the column Sex to either character 'M' or 'F'. Any other value is a violation of this constraint.
- The primary key constraint forces the Emp# column to be unique. If the value of this column is not unique, the employee is not validated.
- In addition to the Emp#, which is used as a primary key and is unique, each Employee row's nickname is also unique. Since a table cannot have two primary keys, the uniqueness of Emp_Nickname is enforced by a different type of constraint in the table, known as a unique constraint.

Referential integrity (RI) constraints restrict the values stored in the database so that they reflect the way the objects of the real world are related to each other. This constraint is inherited from the ERM by the relational systems. In the database, RI is implemented by controlling the way the rows in one table relate to the row(s) in another table. RI in relational systems forces the foreign key values in one table to correspond to a primary key value in another table. Another way of explaining this constraint is to say that in a relational database there exists no foreign key that does not correspond to a primary key. This constraint is enforced by the DBMS automatically.

Figure 3.7 outlines SQL statements that create two tables (Department and Employee) and enforce two referential integrity constraints defined across them. In this figure, we create the Department table first. This table has a clause that specifies the D# column as the primary key of the table. In the Employee table, SSN is defined as the primary key column of the table. MgrSSN, on the other hand, is defined as a foreign key that corresponds to the SSN column of the same table. This column keeps track of the manager of each employee. This constraint is inherited from the ER model, and it corresponds to the fact that in the real world an employee may be the manager of one or more employees. In the relational model, this is implemented by the RI constraint between the SSN and MgrSSN columns of the Employee table. In the relational model, the DBMS has to ensure that every employee in the database has a manager by forcing the value of MgrSSN for a given row to equal one other Employee rows' SSN value. Actually, the DBMS cannot prevent an employee from using its own SSN value for this purpose, but that is not a real issue for use here. We call the employee whose SSN value is used as the MgrSSN value the parent employee. All employees who use the parent's SSN as their MgrSSN are called the children. The DBMS must be told what to do when the SSN for a parent row is changed or deleted. This is specified in the code via 'on update' and 'on delete' clauses. In our example, the DBMS is told that when the parent row is deleted, it should set the value of MgrSSN for all its children to "111111111." The DBMS is also told that changing a SSN value that corresponds to one or more MgrSSN values in the Employee table is not allowed (Restricted). In this case, it does not make sense to allow an Employee's SSN value to be changed.

The DNO column in the Employee table is also defined as a foreign key, which refers to the primary key column of the Department table (D#). This RI keeps track of which department an employee works in. For this RI, we use the 'on update' clause to tell the DBMS to change the column value for DNO in the Employee table if the corresponding value of D# in the Department table changes. The 'on delete' clause of this RI indicates that the requirement is to set the DNO value for an employee to the

```
Create Table Department (D# Integer      not null,
             Name          char(10)      not null,
             Budget        Decimal(12,2) not null,
             Primary Key (D#));
                           (a)

Create Table Employee (SSN char(9)       not null,
             Name          varchar(20)   not null,
             DOB           Date,
             DNO           Integer       default 1,
             MrgSSN        char(9)       not null,
             Primary Key (SSN),
     Foreign Key (MgrSSN) References Employee(SSN)
             On Delete Set lllllllll  On Update Restrict,يمنع
     Foreign Key (DNO) References Department(D#)
             On Delete Set Default   On Update Cascade));
                           (b)
```

Figure 3.7 Example of referential integrity across the Employee and Department tables.

default value when the corresponding department row is deleted from the Department table (i.e., when the department is closed).

Relation constraints as discussed above are typically applied to a single table. The SQL standard allows the use of the SELECT statement in the CHECK constraint to refer to other tables in the same database. To provide for general constraints that can refer to any number of tables, the SQL standard also allows creation of general assertions. Assertions are constraints that are not associated with any individual table. Figure 3.8b illustrates the use of an assertion in conjunction with the database tables shown in Figure 3.8a. As seen from this figure, the general assertion "No_loan_Issue" is applied when a customer requests a new loan from the bank. This assertion prevents the customer being given the loan if the average balance of all the customers' accounts is less than or equal to $5000. Similarly, this assertion ensures that the loan will be denied if the customer already has another loan from the bank (it will not allow the same customer to have more than one loan).

Some commercial databases, like Oracle, do not support the concept of assertions. For these products, explicit semantic integrity constraints have to be used. Explicit constraints are not inherited from the ERM like the other three constraints. We have to either code these constraints into the application programs that use the database or code them into the database using the concept of stored procedures and triggers.

Stored procedures and **triggers** are written programs that use SQL and extensions to the standard SQL—and, as a result, they are vendor specific. These programs are stored in the database as opposed to being compiled and stored as object code outside the database. Stored procedures, similar to any programming language procedures, are programs that accept input parameters, do some work, and can then return values through their output parameters. These procedures can be called from other procedures and/or be activated interactively by the DBMS users. The only difference between a stored procedure and other procedures is that stored procedures cannot request any user interaction once they have started. In other words, we cannot input additional

```
Table Customer(CID, Cname, Addr)
Table Account(A#, CID, bal)
Table Loan(L#, CID, amt)
```

(a)

```
Create Assertion No_loan_Issue
            Check (Select A.CID, Avg(bal)
                   From Account A
                   Group By A.CID
                   Having Avg(bal) > 5000
                   AND
                   Not Exists (
                         Select L.CID
                         From Loan L
                         Where L.CID = A.CID
                              )
                  );
```

(b)

Figure 3.8 An example of an assertion for a sample bank database.

information into the procedure besides what we have given it as input parameters. The general syntax of a stored procedure is given below:

```
Create Procedure Proc_Name (Param_list)
AS
Declaration section;
Begin
     Actions;
End Proc_Name;
```

A trigger is a special kind of stored procedure that is stored in the database. Triggers are an extension to the standard SQL. The only difference between a trigger and a stored procedure is that users cannot call a trigger—only the DBMS can do this. Triggers are run automatically (also known as fired) by the DBMS when certain changes (such as insert, delete, and update operations) are performed on tables of the database. When a table has associated triggers, modification statements performed on the table are monitored by the DBMS and the code of the trigger is fired if the trigger is defined for that action. For example, if we define a trigger for update on the EMP table, the trigger code will run automatically every time the value of a column of EMP is changed. The general syntax of a trigger code is shown below:

```
Create TRIGGER Trigger_Name ON Table FOR operation AS
Declaration section;
Begin
     If Trigger_Condition Exec Proc_Name (Param_list);
     ...
End Trigger_Name;
```

The language that Oracle uses for stored procedures and triggers is called PL/SQL, while SQL Server's language name is T/SQL. Both languages have many similarities but at the same time many differences. For example, Oracle allows for triggers to run either before the modification statement is performed or afterwards. If the trigger runs before the actual modification statement, it is called a **before trigger**; otherwise, it is called an **after trigger**. While Oracle allows both before and after triggers, SQL Server only allows after triggers. Also, Oracle trigger code can be defined to run as many times as the number of rows that are in the target of the modification statement (which is called a **row trigger**) or only once regardless of how many rows are affected (which is called a **statement trigger**). In the case of the SQL Server, only statement triggers are supported. A more detailed explanation of stored procedures and triggers is outside the scope of this book. We encourage interested readers to review any of the many good books written on the subject. In Examples 3.1 and 3.2, we use PL/SQL and T/SQL to write the same stored procedure and trigger for comparison purposes.

Example 3.1 Assume the DEPT and EMP tables are defined as shown in Figure 3.9a in Oracle. We would like to write a stored procedure that will, when given a department

```
EMP (Eno NUMBER(4) not null, Dno NUMBER(4) not null)
DEPT (Deptno NUMBER(4) not null, Mgr varchar(15))
```

(a)

```
1  Create or replace procedure delete_dept (v_dno  INTEGER)
2  Is
3  Begin
4            Delete from DEPT
5            Where deptno = v_dno;
6  End delete_dept;
7  /
```

علي مستو الجدوال

(b)

```
1  create or replace trigger delete_emp
2  after delete on EMP
3  for each row
4  declare
5      v_dno EMP.dno%type;
6      cnt EMP.dno%type;
7  begin
8     select count(*) into cnt
9     From EMP
10     Where EMP.dno = :old.dno;
11     If cnt = 0 then
12     V_dno := :old.dno;
13     End if;
14     delete_dept (v_dno);
15  End delete_emp;
16 /
```

deffrint btween after and After ???????

(c)

Figure 3.9 An example of a stored procedure and a trigger in PL/ SQL.

number (v_dno) as an input parameter, delete that department from the DEPT table. For simplicity, we assume that the department identified by v_dno is stored in the database. Figure 3.9b shows this procedure implemented in PL/SQL. This code simply deletes the department that has its Deptno column equal to the input parameter (see procedure lines 4 and 5). Now, we want to write a trigger that runs when rows are deleted from the EMP table. If the deleted employee is the last employee in the department, the trigger uses the above stored procedure to delete the department from the DEPT table as well. Otherwise, the trigger does not do anything. This trigger is designed to delete the departments when they lose all their employees and have no employees left in them. Figure 3.9c shows the PL/SQL code for this trigger. Since the delete operation may delete more than one row from the EMP table, we need to use a row trigger (see line 3). Since we are using an after trigger, the target employee row is already deleted from the EMP table by the time our code runs. To access the department number for this employee, Oracle maintains the image of the deleted/modified rows in the ":old" record structure (see line 10) during the execution of the trigger code. Here, the field ":old.dno" holds the department number for the row being deleted. The trigger code checks to see if there are any more employees in the department from where the current employee is being deleted (see lines 8 through 13). If this is the case, the department is deleted using the stored procedure in line 14.

Example 3.2 This example is exactly the same as Example 3.1, except that the stored procedure and trigger code are written in T/SQL for SQL Server 2005 DBMS. Note that since T/SQL does not support row triggers, we have to use the concept of a cursor (see lines 5 and 6 in Fig. 3.10) to analyze all the rows being deleted from the EMP table and perform the check for an empty department (see lines 11 through 15). In T/SQL, the table "deleted" holds an image of all the rows that are the target of the delete operation during the execution of the trigger code. We use this table to iterate through all rows that are being deleted. Note that since T/SQL only supports after triggers, during the execution of the trigger code the target rows have already been deleted from the EMP table. As a result, the only way to access the department number for the employees being affected is through the deleted table.

3.4 DISTRIBUTED SEMANTIC INTEGRITY CONTROL

In distributed systems, semantic integrity assertions are basically the same as in centralized systems (i.e., relation, data type, referential, and explicit constraints). When considering a distributed system, semantic integrity control issues become more complicated due to distribution. The first question that we have to answer is, "Where are the semantic integrity rules defined?" When thinking about the location of the rules, we have to be aware of the fact that some semantic integrity rules are local and some are distributed. After we define each rule, we need to consider which tables are affected or referenced by it. When all the tables are deployed in a single database (at a single site), the rule is a local rule; otherwise, it is a distributed rule. We also need to consider where each rule should be defined and enforced. It makes sense to have local rules defined and enforced at the site where all the tables involved are deployed. Deciding where to define and implement the enforcement for distributed (multisite) rules is a much bigger challenge. Let's consider two examples to illustrate these concepts.

```
EMP (Eno DECIMAL(4) not null, Dno DECIMAL(4) not null)
DEPT (Deptno DECIMAL(4) not null, Mgr varchar(15))
```

(a)

```
Create procedure delete_dept @dno  INT
As
Begin
            Delete from DEPT
            Where dno = @dno;
End
go
```

(b)

```
1  Create trigger emp_delete
2  on EMP for delete
3  AS
4  declare @dno int, @value Integer
5  declare C1 Cursor local for
6            select dno from deleted
7  Open C1
8            Fetch next C1 into @dno
9            While @@Fetch_Status = 0
10           Begin
11                        if (select count(*)
12                        From EMP, deleted
13                        Where EMP.dno = @dno) = 0
14                          execute delete_dept @dno
15                        fetch next C1 into @dno
16           end
17 close C1
18 deallocate c1
go
```

(c)

Figure 3.10 An example a stored procedure and a trigger in T/ SQL.

Example 3.3 The RI examples we discussed in Figure 3.7 apply to one database. In a distributed database, we have more than one DBE and might have an RI constraint that spans two or more servers because the tables involved may be deployed at different sites. We might also have an RI constraint that exists in two or more servers because of replication. For this example, assume a distributed database that consists of two servers at Site 1 and Site 2. The Department table, as defined in Figure 3.7a, is stored at Site 1 and the Employee table is stored at Site 2. We still have the same two RI requirements that we had in the previous example for MgrSSN and DNO. These reflect our business requirements and do not have anything to do with the way we have distributed our data. Our problem is with the enforcement of the RI requirement between the Employee and the Department tables. This is a problem, because the two tables are defined on two different DBMSs and neither system can enforce the RI constraint within a foreign system. In other words, when Site 2's DBMS analyzes the 'Create Table Employee. . .' statement, it will complain due to the fact that the Department table does not exist at Site 2—the Department table is defined at Site 1.

Example 3.4 As discussed in Chapter 2, replication is a design technique that attempts to provide more resiliency and availability of data by storing tables and their fragments multiple times. General issues with replication will be discussed in Chapter 7. As far as semantic integrity is concerned, replication creates a unique challenge in enforcing RI. Let's assume that CID is the primary key of the Customer table and that it has been used as a foreign key in the Order table to keep track of customers' orders. Furthermore, assume that both the Customer and Order table have been replicated at Site 1 and Site 2. Let's also suppose that Site 1 is an Oracle database while Site 2 is an IBM DB2 database.

If customer C1 places a new order at Site 1, then Oracle enforces the RI requirement and makes sure that this new order's CID in the Order table matches the Customer's CID. But Oracle does not know (and probably does not care) about the copy of the Order table deployed at Site 2, which is under the control of DB2. As a result, unless the DDBE software replicates this new order in the Order table at Site 2, the replicas of the table will diverge—they are no longer the same. We can also end up with a more interesting and challenging scenario. Let's suppose that at the same time that customer 100 places a new order for item 13 at Site 1, the DBA at Site 2 deletes item 13 from its database. The referential integrity of the Oracle and DB2 databases locally is enforced and the systems individually are consistent. But, as far as the overall distributed database is concerned, the database is mutually inconsistent. That is because at Site 1 we have an order for an item that does not exist at Site 2. These and similar issues related to semantic integrity constraints have to be addressed by the replication control software (see Chapter 7). These examples indicate many challenges involved in the declaration and enforcement of semantic integrity constraints in a distributed system.

The following outlines the additional challenges in distributed semantic integrity control:

- Creation of multisite semantic integrity rules
- Enforcement of multisite semantic integrity rules
- Maintaining mutual consistency of local semantic integrity rules across copies
- Maintaining consistency of local semantic integrity rules and global semantic integrity rules

It should be obvious that these challenges are due to database fragmentation, replication, and distribution. Obviously, the more a database is fragmented and/or replicated, the more overhead will be involved in creation and enforcement of the integrity rules. The main component of this overhead is the cost of communication between sites. An attempt to minimize the cost of communication can reduce the overall overhead associated with the creation and enforcement of integrity rules in a distributed database system. One approach for minimizing communication cost is to try to distribute a database in such a way that does not create multisite semantic integrity rules. For example, we should place the EMP and DEPT tables as shown in Figure 3.7a at the same site due to the existence of the RI rule between them. As another example, we could avoid vertical fragmentation of any table, since otherwise we would need to enforce relation constraints across the vertical fragments for each fragmented table.

Much of the work in semantic integrity area has been focused on the specification of the rules [Stonebreaker74] [Eswaran76] [Machgeles76] [McLeod79] as opposed to how

the rules are validated [Eswaran76] [Stonebreaker76] [Hammer78]. Badal argues that, for a given distribution design, when we enforce the semantic integrity rules [Badal79] it has a big impact on the communication cost. According to Badal, semantic integrity rules for a transaction can be enforced when the transaction is compiled (compile time validation), during its execution (run time validation), or after the transaction has finished executing (postexecution time validation). In the following subsections, we briefly discuss these three approaches. For each approach, we calculate the cost of the approach, based on the number of required messages.

3.1 Compile Time Validation

In this approach to validation, transactions are allowed to run only after all the semantic integrity (SI) constraints have been validated. For this approach to work, SI data items need to be locked so that during validation and afterward, during transaction execution, they are not changed. It is easy to see that compile time validation is simple to implement and does not incur any cost for abort operations since we only run transactions when they are validated—transactions that violate the SI rule do not run at all. On the other hand, to implement this approach, all the constraint data items need to be locked for the duration of validation and transaction execution. After a transaction has been validated, it starts its execution. Note that a transaction that has been validated may still be rolled back due to concurrency control issues (such as locking and deadlock) or due to conflicts with other concurrent transactions.

3.4.2 Run Time Validation

In this validation scheme, there is no need to hold transactions back until they are validated (like we did in compile time validation). Here, transactions are validated during execution. When there is a need for validation, all the data items involved are locked and the transaction is validated. If semantic integrity rules are violated, the transaction is rolled back. There is a larger overhead associated with this approach, as compared to compile time validation, since invalid transactions need to be rolled back. On the other hand, the duration for which the constraint data items need to be locked is shorter. Once a transaction has been validated, it can commit.

3.4.3 Postexecution Time Validation

In this approach, validation is performed after the execution of the transaction but just before the transaction is committed. If validation fails, transaction will be aborted. It should be obvious that the cost of abort in this case is the highest since validation is performed last. Consequently, the lock duration for the data items with which a transaction is working is also long.

3.5 COST OF SEMANTIC INTEGRITY ENFORCEMENT

In centralized system, the cost of enforcing SI consists of the cost associated with accessing SI data items, locking these data items, calculating assertions associated with them, and then releasing the locks. This cost is dominated by the database access

4

QUERY OPTIMIZATION

In this chapter, we provide an overview of query processing with the emphasis on optimizing queries in centralized and distributed database environments. It is a well-documented fact that for a given query there are many evaluation alternatives. The reason for the existence of a large number of alternatives (**solution space**) is the vast number of factors that affect query evaluation. These factors include the number of relations in the query, the number of operations to be performed, the number of predicates applied, the size of each relation in the query, the order of operations to be performed, the existence of indexes, and the number of alternatives for performing each individual operation—just to name a few. In a distributed system, there are other factors, such as the fragmentation details for the relations, the location of these fragments/tables in the system, and the speed of communication links connecting the sites in the system. The overhead associated with sending messages and the overhead associated with the local processing speed increase exponentially as the number of available alternatives increases. It is therefore generally acceptable to merely try to find a "good" alternative execution plan for a given query, rather than trying to find the "best" alternative.

A query running against a distributed database environment (DDBE) will have to go through two types of optimization. The first type of optimization is done at the global level, where communication cost is a prominent factor. The second type of optimization is done at the local level. This is what each local DBE performs on the fragments that are stored at the local site, where the local CPU and, more importantly, the disk input/output (I/O) time are the main drivers. Almost all global optimization alternatives ignore the local processing time. When these alternatives were being developed, it was believed that the communication cost was a more dominant factor than the local processing cost. Now, it is believed that both the local query cost and the global communication cost are important to query optimization.

Distributed Database Management Systems by Saeed K. Rahimi and Frank S. Haug

Suppose we have two copies of a relation at two different servers, where the first server is a lot faster than the second server, but at the same time, the connection to the first server is a lot slower than the connection to the second server (perhaps we are closer to the second server). An optimization strategy that only considered communication cost would choose the second server to run the local query. This will not necessarily be the best strategy, due to the speed of the chosen (second) server. The overall time to run a query in a distributed system consists of the time it takes to communicate local queries to local DBEs; the time it takes to run local query fragments; the time it takes to assemble the data and generate the final results; and the time it takes to display the results to the user. Therefore, to study distributed query optimization, we need to understand how a query is optimized both locally and globally.

In this chapter, we introduce the architecture of the query processor for a centralized system first. We then analyze how a query is processed optimally, discussing the optimization techniques in a centralized system. The optimization of queries in a distributed system is explained last. We introduce a simple database that we use in our examples. We will also provide a brief introduction to **relational algebra (RA)** in this chapter, since most commercial database systems use this language as an internal representation of SQL queries.

4.1 SAMPLE DATABASE

We will use a small database representing a bank environment for our examples. This database has five relations: Customer, Branch, Account, Loan, and Transaction. In this database, customers, identified by CID, open up accounts and/or loans in different branches of the bank that are located in different cities. This is indicated by the CID and BNAME foreign keys in the Account and Loan relations. Customers also run transactions against their accounts. This is shown in the Transaction relation by the combined foreign key "(CID, A#)." Later in this chapter, when discussing query optimization alternatives, we will specify the statistics for this database. The following shows the relations of our example bank database.

```
CUSTOMER (CID, CNAME, STREET, CCITY);
BRANCH (BNAME, ASSETS, BCITY);
ACCOUNT (A#, CID, BNAME, BAL);
LOAN (L#, CID, BNAME, AMT);
TRANSACTION (TID, CID, A#, Date, AMOUNT);
```

4.2 RELATIONAL ALGEBRA

Since the introduction of the relational model by Codd in 1970 [Codd70], two classes of languages have been proposed and implemented to work with a relational database. The first class is called **nonprocedural** and includes **relational calculus** and **Quel**. The second class is known as **procedural** and includes relational algebra and the **Structured Query Language** (**SQL**) [SQL92]. In procedural languages, the query directs the DBMS on **how** to arrive at the answer. In contrast, in a nonprocedural language, the query indicates **what** is needed and leaves it to the system to find the

process for arriving at the answer. Although it sounds easier to tell the system what is needed instead of how to get the answer, nonprocedural languages are not as popular as procedural languages. As a matter of fact, SQL (a procedural language) is the only widely accepted language for end user interface to relational systems today.

To start our query processing discussion, we will make the assumption that user requests are entered into the system as SQL statements. This is because, as we mentioned before, one of the goals of a distributed database management system is to provide a standards-based, uniform, high-level language interface to all the data that is stored across the distributed system. SQL is typically used as such a high-level language interface. Even though SQL is an accepted and popular interface for end users, it does not lend itself nicely to internal processing. Perhaps the most problematic aspect of SQL is its power in representing complex queries easily at a very high level without specifying how the operations should be performed. That is why most commercial database systems use an internal representation based on relational algebra that specifies the ordering of different operations within the query. Therefore, to understand how SQL queries are processed, we need to understand how their equivalent relational algebra commands work.

Since the detailed discussion of RA is outside the scope of this book, in this chapter we will only provide an overview of those RA operations that are of interest to query optimization. We recommend interested readers look at the discussion of the relational data model in Section 11.1 for a brief review of the terminology, and see [Codd70] for a more detailed review of the concepts and operations.

For the remainder of this section we will use the following notations:

- R and S are two relations.
- The number of tuples in a relation is called the cardinality of that relation.
- R has attributes a1, a2, . . . , an and has cardinality of K.
- S has attributes b1, b2, . . . , bm and has cardinality of L.
- r is a tuple in R and is shown as r[a1, a2, . . . , an].
- s is a tuple in S and is shown as s[b1, b2, . . . , bm].

4.2.1 Subset of Relational Algebra Commands

Relational algebra (**RA**) supports unary and binary types of operations. Unary operations take one relation (table) as an input and produce another as the output. Binary operations take two relations as input and produce one relation as the output. Note that regardless of the type of operation, the output is always a relation. This is an important observation since the output of one operation is usually fed as an input into another operation in the query. RA operators are divided into **basic operators** and **derived operators**. Basic operators need to be supported by the language compiler since they cannot be created from any other operations. Derived operators, on the other hand, are optional since they can be expressed in terms of the basic operators. Greek symbols are sometime used to represent the RA operators in many textbooks (see Table 4.1).

In this book, we will use the following notation instead:

- SL represents the relational algebra SELECT operator.
- PJ represents the relational algebra PROJECT operator.

TABLE 4.1 Symbols Often Used to Represent Relational Algebra Operators

Symbol	Name	RA Operator
σ	Sigma	Select
π	Pi	Project
$\bowtie$	Bowtie	Cross product or join

- JN represents the relational algebra JOIN operator.
- NJN represents the relational algebra natural JOIN operator.
- UN represents the relational algebra UNION operator.
- SD represents the relational algebra natural SET DIFFERENCE operator.
- CP represents the relational algebra CROSS PRODUCT operator.
- SI represents the relational algebra SET INTERSECT operator.
- DV represents the relational algebra DIVIDE operator.

4.2.1.1 Relational Algebra Basic Operators Basic operators of RA are SL, PJ, UN, SD, and CP. In the following subsections, we briefly describe these operators.

Select Operator in Relational Algebra The select operator returns all tuples of the relation whose attribute(s) satisfy the given predicates (conditions). If no condition is specified, the select operator returns all tuples of the relation. For example, "$SL_{bal=1200}$ (Account)" returns all accounts that have a balance of $1200. The result is a relation with four attributes (since the Account relation has four attributes) and as many rows as the number of accounts with a balance of exactly $1200. The predicate "bal = 1200" is a simple predicate. We can use "AND," "OR," and "NOT" to combine simple predicates, making complex predicates. For example, we can find the accounts with a balance of $1200 at branch "Main" using the select expression, "$SL_{bal=1200\ AND\ Bname='Main'}$ (Account)."

Project Operator in Relational Algebra The project operator returns the values of all attributes specified in the project operation for all tuples of the relation passed as a parameter. In a project operation, all rows qualify but only those attributes specified are returned. For instance, "PJ $_{Cname,Ccity}$ (Customer)" returns the customer name and the city where the customer lives for each and every customer of the bank.

COMBINING SELECT AND PROJECT We can combine the select and project operators in forming complex RA expressions that not only apply a given set of predicates to the tuples of a relation but also trim the attributes to a desired set. For example, assume we want to get the customer ID and customer name for all customers who live in Edina. We can do this by combining the SL and the PJ expressions as "PJ $_{CID,\ Cname}$ (SL $_{Ccity='Edina'}$ (Customer))." Note that operator precedence is enforced by parentheses. In this example, the innermost expression is the SL operation and is carried out first. This expression returns all customers who live in Edina. Subsequently, the PJ operator trims the results to only CID and Cname for those customers returned from the SL operation.

Union Operator in Relational Algebra Union is a binary operation in RA that combines the tuples from two relations into one relation. Any tuple in the union is in the first relation, the second relation, or both relations. In a sense, the union operator in RA behaves the same way that the addition operator works in math—it adds up the elements of two sets. There are two compatibility requirements for the union operation. First, the two relations have to be of the same degree—the two relations have to have the same number of attributes. Second, corresponding attributes of the two relations have to be from compatible domains.

The following statements are true for the union operation in RA:

- We cannot union relations "R(a1, a2, a3)" and "S(b1, b2)" because they have different degrees.
- We cannot union relations "R(a1 char(10), a2 Integer)" and "S(b1 char(15), b2 Date)" because the a2 and b2 attributes have different data types.
- If relation "R(a1 char(10), a2 Integer)" has cardinality K and relation "S(b1 char(10), b2 Integer)" has cardinality L, then "R UN S" has cardinality "K + L" and is of the form "(c1 char(10), c2 Integer)."

Suppose we need to get the name and the address for all of the customers who live in a city named "Edina" or "Eden Prairie." To find the results, we first need to create a temporary relation that holds Cname and Ccity for all customers in Edina; then we need to repeat this for all the customers in Eden Prairie; and finally, we need to union the two relations. We can write this RA expression as follows:

```
PJCID, Cname (SLCcity = 'Edina' (Customer))
UN
PJCID, Cname (SLCcity = 'Eden Prairie' (Customer))
```

The union operator is commutative, meaning that "R UN S = S UN R." Also, the union operator is associative, meaning that "R UN (S U P) = (R UN S) UN P." Applying associativity and commutativity properties to union, we end up with 12 different alternatives for union of the three relations R, S, and P. We will leave it as an exercise for the reader to enumerate all the possible alternatives for the union of three relations.

Set Difference Operator in Relational Algebra Set difference (SD) is a binary operation in RA that subtracts the tuples in one relation from the tuples of another relation. In other words, SD removes the tuples that are in the intersection of the two relations from the first relation and returns the result. In "S SD R," the tuples in the set difference belong to the S relation but do not belong to R. Set difference is an operator that subtracts the elements of two sets. In a sense, the set difference operator in RA behaves the same way that the subtraction operator works in math. There are again two compatibility requirements for this operation. First, the two relations have to be the same degree, and second, the corresponding attributes of the two relations have to come from compatible domains.

Assume we need to print the customer ID for all customers who have an account at the Main branch but do not have a loan there. To do this, we first form the set of all customers with accounts at the Main branch and then subtract all the customers with

a loan at the Main branch from that set. This excludes the customers who are in the intersection of the two sets (those who have both an account and a loan at the Main branch) leaving behind the desired customers. The RA expression for this question is written as

$$PJ_{CID}\ (SL_{Bcity\ =\ 'Main'}\ (Account))$$
$$SD$$
$$PJ_{CID}\ (SL_{Bcity\ =\ 'Main'}\ (Loan))$$

Note: The SD operator is not commutative, that is, "R SD S $\neq$ S SD R." That is because the left-hand side of the inequality returns tuples in R that are not in S, while the right-hand side returns tuples in S that are not in R. Note, however, that unlike the union operator, the SD operator is not associative, meaning that "R SD (S SD P) $\neq$ (R SD S) SD P."

Cartesian Product Operator in Relational Algebra Cartesian product (CP), which is also known as cross product, is a binary operation that concatenates each and every tuple from the first relation with each and every tuple from the second relation. CP is a set operator that multiplies the elements of two sets. In a sense, the CP operator in RA behaves the same way that the multiplication operator works in math. This operation is hardly used in practice, since it produces a large number of tuples—most of which do not contain any useful information. "R CP S" is a relation with L*K tuples and each tuple is of the form "[a1, a2, . . ., an, b1, b2, . . ., bm]." For example, assume we have 1000 accounts and 200 loans in the bank. The cross product of the account and loan relations, written as "Account CP Loan," will have 8 attributes and as many as 200,000 tuples. Table 4.2 shows some of the tuples in the "Account CP Loan" results.

As seen from the sample tuples in this relation, account information for account number 100, for customer 111, has been concatenated with all the loans in the bank. Although this is a valid relation as far as the relational model is concerned, all the rows except for the ones that have equal values for the two CID attributes are useless (the tuple with "L# = 167" is valid, but the other tuples are not valid). Therefore, only tuple number 2, where the account and loan information for customer 111 have been concatenated, contains meaningful results. We will discuss how to eliminate the tuples that do not contain meaningful information when we discuss the join operator.

4.2.1.2 Relational Algebra Derived Operators In addition to the basic operators in RA, the language also has a set of derived operators. These operators are called "derived" since they can be expressed in terms of the basic operators. As a result, they are not required by the language, but are supported for ease of programming. These

TABLE 4.2 Partial Cartesian Products Results

A#	CID	Bname	Bal	L#	CID	Bname	Amt
100	111	Main	1000	212	312	Main	20000
100	111	Main	1000	167	111	Main	5000
100	111	Main	1000	435	217	Main	120000
100	111	Main	1000	900	222	Edina	63000

operators are SI, JN (NJN), and DV. The following sections represent an overview of these operators.

Set Intersect Operator in Relational Algebra Set intersect (SI) is a binary operator that returns the tuples in the intersection of two relations. If the two relations do not intersect, the operator returns an empty relation. Suppose that we need to get the customer name for all customers who have an account and a loan at the Main branch in the bank. Considering the set of customers who have an account at the Main branch and the set of customers who have a loan at that branch, the answer to the question falls in the intersection of the two sets and can be expressed as follows:

$$PJ_{Cname}\ (SL_{Bcity\ =\ 'Main'}\ (Account))$$
$$SI$$
$$PJ_{Cname}\ (SL_{Bcity\ =\ 'Main'}\ (Loan))$$

SI operation is associative and commutative. Therefore, "R SI S = S SI R" and "R SI (S SI P) = (R SI S) SI P." As mentioned before, SI is a derived operator. That is because we can formulate the intersection of the two relations, "R SI S" as "R SD (R SD S)" by the successive application of the SD operator. We leave the proof as an exercise for the reader. *Hint*: Use a Venn diagram [Venn1880].

Join Operator in Relational Algebra The join (JN) operator in RA is a special case of the CP operator. In a CP operation, rows from the two relations are concatenated without any restrictions. However, in a JN, before the tuples are concatenated, they are checked against some condition(s). JN is a binary operation that returns a relation by combining tuples from two input relations based on some specified conditions. These operations are known as **conditional joins**, where conditions are applied to the attributes of the two relations before the tuples are concatenated. For instance, the result of "R $JN_{a2 > b2}$ S" is a relation with "$<= L * K$" tuples and each tuple is in the form "[a1, a2, . . . , an, b1, b2, . . . , bm]," satisfying the condition "$a2 > b2$." One popular join condition is to force equality on the values of the attributes of the two relations. These types of joins are known as **equi-joins**. The expression "R $JN_{a2=b2}$ S" is a join that returns a relation with "$<= L * K$" tuples and each tuple is in the form "[a1, a2, . . . , an, b1, b2, . . . , bm]," satisfying the condition "$a2 = b2$."

In addition to these, RA supports the concept of **natural join**, where equality is enforced automatically on the attributes of the two relations that have the same name. For example, consider relations "Account(A#, CID, Bname, bal)" and "Branch(Bname, Bcity, Budget)." Although we can join these relations enforcing equality on any of their two attributes, the natural way to join them is to force equality on the Bname in Branch and the Bname in Account. A careful reader notices that this type of join forces equality on the value of the primary key with the value of the foreign key, resulting in groups of accounts for each branch. This join is written as "Branch $JN_{Branch.Bname=Account.Bname}$ Account." When performing joins of this nature, Relational Algebra knows that the values of attributes "Branch.Bname" and "Account.Bname" are the same, and therefore, showing both columns is not necessary. That is why RA drops one of the columns and only displays one. For our example, this type of join is called a natural join and is shown as "Branch NJN Account" knowing that equality is forced automatically on the

Bname attributes. Note that any natural join operation reduces to a Cartesian products operation if the two relations have no attributes in common.

Figure 4.1 depicts the union, SD, and SI operators in RA using Venn diagrams. The shaded areas for UN and SD and the cross-hatched area for SI represent the answer for each operator.

Divide Operator in Relational Algebra Divide (DV) is a binary operator that takes two relations as input and produces one relation as the output. The DV operation in RA is similar to the divide operation in math. We will use an example to show how the divide operation works in relational algebra. Assume we want to find all customers who have an account at all of the branches located in the city MPLS (each and every branch). As a city, MPLS has multiple branches located in it. Suppose branches in MPLS are Main, Nicollet, and Marquette. We are looking for all customers who have at least one account in **every one of these three branches**. For instance, Jones who has one account in Main and one account in Nicollet is not part of the answer, while Smith who has an account in Main, an account in Nicollet, and an account in Marquette is part of the answer. Jones would be part of the answer if the question were, "Find customers who have an account at ANY branch in MPLS." The key difference between these two questions is "all" versus "any." To get the answer, we need to form two sets of tuples. The first is the set of all customers and the branches in which they have an account. The second set is the set of all branches in MPLS. A customer is part of the answer if for that customer the set of customers' branches **contains** the set of branches in MPLS. The operation that performs these steps is the division operation. Figure 4.2 depicts the two sets and the answer after the divide operation.

The expression "S = PJ bname ($SL_{bcity = 'MPLS'}$ (Branch))" represents the set of all branch names in MPLS. The expression "R = $PJ_{cname, bname}$ (Account NJN Customer)"

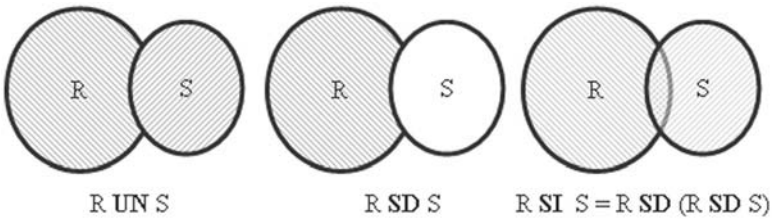


Figure 4.1 Set operations in relational algebra.

R

cname	bname
Smith	b1
Rahimi	b3
Jones	b2
Rahimi	b2
Smith	b2
Rahimi	b1
Love	b4

DV

S

bname
b1
b2

=

Answer

cname
Smith
Rahimi

Figure 4.2 Set divide operation example.

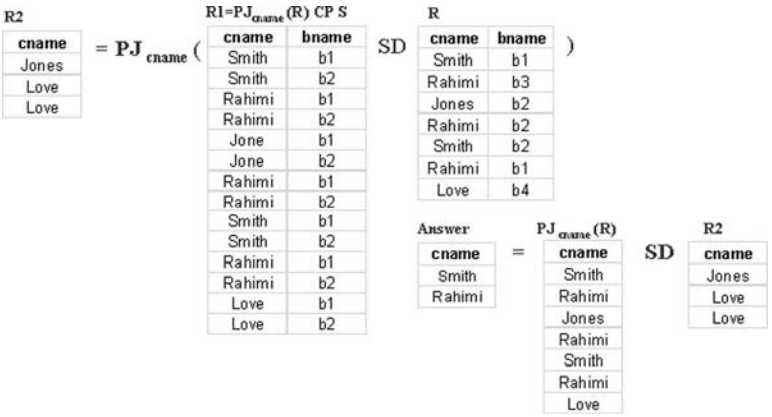


Figure 4.3 Divide operation using the basic RA operations.

represents the set of all customer names and their branch names for all customers in the bank. Hence, the answer can be arrived at by the following:

$PJ_{cname,\ bname}$ (Account NJN Customer)
DV
PJ_{bname} ($SL_{bcity\ =\ 'MPLS'}$ (Branch))

The following explains what the DV operator actually does:

- First, the DV operation performs a group-by on the Cname attribute of the first relation, which results in a set of branch names for each customer.
- Then, it checks to see if the set of Bname values associated with every unique value of Cname is the same set or a superset of the set of Bname values from the second relation. If it is (the same set or superset), then the customer identified by that Cname is part of the division result.

Since DV is a derived operation it can be expressed using the base RA operations expressed as follows:

R1 = PJ_{cname}(R) CP S
R2 = PJ_{cname} (R1 SD R)
R(cname, bname) DV S(bname) = PJ_{cname} (R) SD R2

We have shown the results of these steps in Figure 4.3 for the bank example.

4.3 COMPUTING RELATIONAL ALGEBRA OPERATORS

In the previous section, we discussed the use of some relational algebra operators. In this section, we briefly explain how each operator can be processed. As we will see in this section, an operator in relational algebra can be computed in many different ways

Assuming we have 201 buffers for the sort, relation R can be sorted in

$$\begin{aligned} BW\_Total\_cost &= 2 * 1000 * (1 + \lceil Log_{200} 5 \rceil) \\ &= 2 * 1000 * (1 + 1) \\ &= 4000 \text{ disk I/Os} \end{aligned}$$

The typical number of buffers allocated to the sort in today's DBMSs is 257. This allows for a 256-way sort–merge and requires only three passes for a relation with 1,000,000 pages and only four passes for a relation of 1,000,000,000 pages!

Merge Merge is the last step in joining two relations using the sort–merge approach. To merge, each page of the sorted relations must be brought into memory and then written out. The total cost of merging R and S, for example, is "2 * (M + N)." The total cost of the sort–merge join for R and S using B buffers is

$$\begin{aligned} Total\_cost = \quad & 2 * M * (1 + \lceil Log_{B-1} \lceil M / B \rceil \rceil) \\ & + 2 * N * (1 + \lceil Log_{B-1} \lceil N / B \rceil \rceil) \\ & + 2 * (M + N) \end{aligned}$$

4.3.2.3 Hash-Join The last alternative access path we will consider in this section is the hash-join. The hash-join has been gaining a lot of popularity in today's DBMSs. The hash-join approach consists of two phases: the **partitioning** phase and the **probing** phase. In the partitioning phase, R and S are each broken up (partitioned) into two separate collections of nonoverlapping subsets (partitions). A tuple is assigned to a particular partition by using the same hash function for both relations. When partitioning R, attribute "a" is passed to the hash function, while the partitioning of S passes attribute "b." Since we use the same hash function for both relations in the partitioning phase, the matching tuples from both relations, if any, end up in the buckets with the same address. For example, if all key values between 100 and 200 are hashed to partition 5 of R, then all tuples with the same key value range in S will also be hashed into partition 5 of S. In the probing phase of the join, tuples in a partition of R are only compared to the tuples in the corresponding partition of S. There is no need to compare the tuples in partition 5 of R, for example, with tuples in partition 7 of S. This is a very important observation that gives a hash-join its speed. Of course, we still have to spend the time it takes to partition the two tables based on the values of the join attribute. But, once we have done that, the probing phase can be carried out very quickly.

The cost of hash-join is the total cost of partitioning both relations and the cost of probing. In the partitioning phase, each relation is scanned and is written back to the disk. Therefore, the partitioning phase cost is "2 * (M + N)." In the probing phase, each relation is scanned once again. Hence, the total cost of a hash-join is "3 * (M + N)." Using the statistics we used earlier in this section, the hash-join of R and S will take "3 * (1000 + 500)" or 4500 disk I/Os or 40.5 seconds. Compared to the nested-loop join access path, this is a much smaller cost!

4.4 QUERY PROCESSING IN CENTRALIZED SYSTEMS` "Reading"

The goal of the query processing subsystem of a DBMS should be to minimize the amount of time it takes to return the answer to a user's query. Obviously, the response

time metric is the most important to the user. However, there are other cost elements that the system is concerned with, which do not necessarily result in the best response time. For example, the system may decide to optimize the amount of resources it uses to get the answer to a given user's query or the answers to queries requested by a group of users. In other cases, we may try to maximize the throughput for the entire system. This may translate into a "reasonable" response time for all queries rather than focusing on the response time for a specific query. These goals are sometime contradictory and do not always mean the fastest response time for a given user or group of users.

In a centralized system, the goals of the query processor may include the following:

- Minimize the query response time.
- Maximize the parallelism in the system.
- Maximize the system throughput.
- Minimize the total resources used (amount of memory, disk space, cache, etc.).
- Other goals?

The system might be unable to realize all of these goals. For example, minimizing the total resource usage may not yield minimum query response time. It is understood that minimizing the amount of memory allocated to sorting relations can have a direct impact on how fast a relation can be sorted. Faster sorting of the relations speeds up the total amount of time needed to join two relations using the **sort–merge** (see Section 4.3.2.2) strategy. The more **memory pages** (**memory frames)** allocated to the sort process the faster the sort can be done, but since total physical memory is limited, increasing the size of sort memory decreases the amount of memory that can be allocated to other data structures, temporary table storage in memory, and other processes. In effect, this **may increase** the query response time.

The center point of any query processor in a centralized or distributed system is the **data dictionary** (DD) (the **catalog**). In a centralized system, the catalog contains dictionary information about tables, indexes, views, and columns associated with each table or index. The catalog also contains statistics about the structures in the database. A system may store the number of pages used by each relation and indexes, the number of rows per page for a given relation, the number of unique values in the key columns of a given relation, the types of keys, the number of leaf index pages, and so on. In a distributed system, the catalog stores additional information that pertains to the distribution of the information in the system. Information on how relations are fragmented, the location of each fragment, the speed of communication links connecting sites, the overhead associated with sending messages, and the local CPU speed are all examples of the details the catalog may contain in a distributed system.

4.4.1 Query Parsing and Translation

As shown in Figure 4.4, the first step in processing a query is parsing and translation. During this step, the query is checked for syntax and correctness of its data types. If this check passes, the query is translated from its SQL representation to an equivalent relational algebra expression.

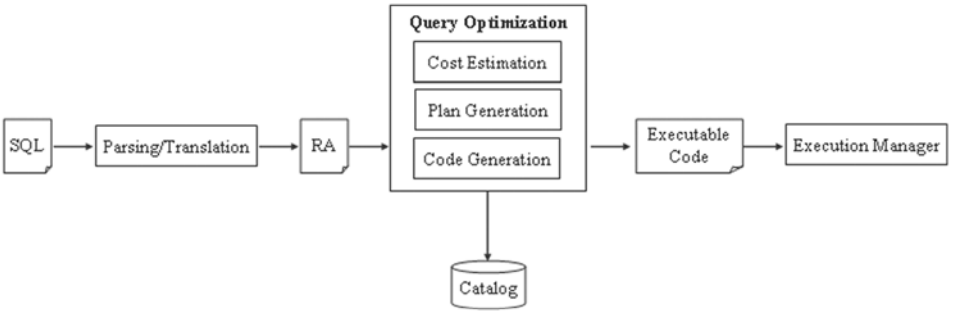


Figure 4.4 Query processing architecture of a DBE.

Example 4.1 Suppose we want to retrieve the name of all customers who have one or more accounts in branches in the city of Edina. We can write the SQL statement for this question as

```
Select c.Cname
From   Customer c, Branch b, Account a
Where  c.CID   = a.CID
AND    a.Bname = b.Bname
AND    b.Bcity = 'Edina';
```

There are two join conditions and one select condition (known as a filter) in this statement. The relational algebra (RA) expression that the parser might generate is shown below:

$$PJ_{cname}\ (SL_{Bcity = \text{'Edina'}}\ (\text{Customer CP (Account CP Branch)}))$$

The DBMS does not execute this expression as is. The expression must go through a series of transformations and optimization before it is ready to run. The query optimizer is the component responsible for doing that.

4.4.2 Query Optimization

There are three steps that make up query optimization. These are cost estimation, plan generation, and query plan code generation. In some DBMSs (e.g., DB2), an extra step called "Query Rewrite" is performed before query optimization is undertaken. In query rewrite, the query optimizer rewrites the query by eliminating redundant predicates, expanding operations on views, eliminating redundant subexpressions, and simplifying complex expressions such as nesting. These modifications are carried out regardless of database statistics [Pirahesh92]. Statistics are used in the optimization step to create an optimal plan. Again, an optimal plan may not necessarily be the best plan for the query.

The RA expression for Example 4.1 does not run efficiently, since forming Cartesian products of the three tables involved in the query produces large intermediate relations. Instead, join operators are used and the expression is rewritten as

$$PJ_{cname}\ (\text{Customer NJN Account})\text{NJN}\ (\text{Account NJN}\ (SL_{Bcity = \text{'Edina'}}(\text{Branch})))$$

This expression can be refined further by eliminating the redundant joining of Account relation as

$$PJ_{cname}\ (Customer\ NJN\ (Account\ NJN\ (SL_{Bcity='Edina'}\ (Branch))))$$

As we will see later, there are other equivalent expressions that can also be used. All available alternatives are evaluated by the query optimizer to arrive at an optimal query expression.

4.4.2.1 Cost Estimation Given a query with multiple relational algebra operators, there are usually multiple alternatives that can be used to express the query. These alternatives are generated by applying the associative, commutative, idempotent, distributive, and factorization properties of the basic relational operators [Ceri84].

These properties are outlined below (the symbol "≡" stands for equivalence):

- Unary operator (Uop) is commutative:

  $$Uop1(Uop2(R)) \equiv Uop2(Uop1(R))$$

  For example,

  $$SL_{Bname = 'Main'}\ (SL_{Assets > 12000000}\ (Branch)) \equiv SL_{Assets > 12000000}\ (SL_{Bname = 'Main'}\ ((Branch))$$

- Unary operator is idempotent:

  $$Uop((R)) \equiv Uop1(Uop2((R))$$

  For example,

  $$SL_{Bname = 'Main'\ AND\ Assets > 12000000}\ (Branch) \equiv SL_{Bname = 'Main'}\ (SL_{Assets > 12000000}\ (Branch)$$

- Binary operator (Bop) is commutative except for set difference:

  $$R\ Bop1\ S \equiv S\ Bop1\ R$$

  For example,

  $$Customer\ NJN\ Account \equiv Account\ NJN\ Customer$$

- Binary operator is associative:

  $$R\ Bop1\ (S\ Bop2\ T) \equiv (R\ Bop1\ S)\ Bop2\ T$$

  For example,

  $$Customer\ NJN\ (Account\ NJN\ Branch) \equiv (Customer\ NJN\ Account)\ NJN\ Branch$$

5

CONTROLLING CONCURRENCY

In this chapter, we will discuss the concepts of transaction management and database consistency control. We start by defining the terms we will use in the rest of the chapter. We then focus on transaction management as it relates to **online transaction processing** (**OLTP**) systems. The concurrency control service is the DBE service that is responsible for consistency of the database. In a nutshell, it controls the operations of multiple, concurrent transactions in such a way that the database stays consistent even when these transactions conflict with each other. Approaches to **concurrency control** are explained next. We introduce concurrency control for centralized DBE first and then consider distributed DBE. In Section 5.3, we formalize the algorithms for concurrency control in a centralized DBE. In Section 5.4, we expand the concurrency control for a centralized DBE to cover the issues in concurrency control for distributed DBEs.

5.1 TERMINOLOGY

Before we discuss how concurrency control applies to the centralized and distributed databases, it is important to understand the terminology that we are using to describe the different approaches and issues involved. In particular, we need to have a better understanding of exactly what a database and a transaction are.

5.1.1 Database

A **database** is a collection of data items that have a name and a value. The set $D\{i_1, i_2, \ldots, i_N\}$ represents a database with N data items. Some of these data items must have a value, they are NOT NULL, and some may have a value, they are NULL. Although this definition of the database seems simplistic, in reality it is comprehensive and can represent relational databases, object-oriented databases, hierarchical databases,

Distributed Database Management Systems by Saeed K. Rahimi and Frank S. Haug

NAME	ADDRESS	CITY
Paul Jones	1437 Washington	St. Paul
Cindy Smith	2645 France	Edina

Figure 5.1 A generic database example.

network databases, spreadsheet databases, and flat file databases. Even though we assume each data item has a name, in reality, all we are saying is that every item is accessible and/or addressable. Figure 5.1 depicts an example of a database. According to our definition, there are six data items in this database.

To a person familiar with an Excel spreadsheet, this database represents a sheet with two rows and three columns. Each cell is addressable by the combination of a row and column address. The value "2645 France" represents the address for Cindy Smith. Now assume this is an object-oriented database. For an object-oriented database, "2645 France" presents the address property for the Cindy Smith object while "Paul Jones" represents a name property for the Paul Jones object. In a relational database, this example represents a table with two rows and three columns. In a relational database, the value, "2645 France" is associated with the address column of Cindy Smith's row.

5.1.1.1 Database Consistency Each data item in the database has an associated correctness assertion. For example, the social security number for an employee must be a unique value, age for an employee needs to be a positive number, an employee must work for one and only one department, the balance of an account needs to be positive and more than $100, and so on. These are all examples of assertions in a database. A database is said to be consistent if and only if the correctness criteria for all the data items of the database are satisfied.

5.1.2 Transaction

A **transaction** is a collection of operations performed against the data items of the database. There have been many references to the **ACID** properties (**atomicity, consistency, isolation**, and **durability**) of a transaction in the literature. Instead of simply repeating these properties, we will use an example to explain a transaction's properties.

In order to know what a transaction does, we need to know when it starts and when it ends. We do that by delineating the boundaries of the transaction either explicitly or implicitly. For our discussions, we will indicate the transaction boundaries using **Begin_Tran** and **End_Tran**. Note that when working interactively with the DBMS we may not need to use Begin_Tran and End_Tran. That is because, in most DBMSs, either every user's operation is considered a transaction or the entire user's session is considered as one large transaction. Similarly, when a programmer implements a transaction, the use of Begin_Tran and End_Tran might be required. The syntax and precise mechanism used to perform these operations varies, depending on the DBMS, the database/transaction API, and the programming language/platform being used to implement the program (we will look at some different mechanisms in Chapters 15–19). In the body of the transaction, one can find some control statements, some read operations, some write operations, some calculations, zero or more aborts, and finally zero or one commit operation.

Example 5.1 This example shows pseudocode (an implementation independent form of an algorithm) for a transaction that transfers $500 from account X to account Y.

```
Begin_Tran T1:
     Read (account X's balance) into X
     If account X not found Then
          Print 'Invalid source account number'
          Abort
     End If
     Read (account Y's balance) into Y
     If account Y not found Then
          Print 'Invalid target account number'
          Abort
     End If
     Calculate X = X - 500
     If X < 0 Then
          Print 'Insufficient funds'
          Abort
     End If
     Calculate Y = Y + 500
     Write (account X's balance) with X
     Write (account Y's balance) with Y
     Ask user if it is OK to continue
     If Answer is "Yes" Then
          Commit
               Else
          Abort
     End If
End_Tran T1;
```

To simplify our discussion when working with transactions, we will disregard the calculations performed and only focus on the read and write operations within the transaction. No information is lost when we do this. That is because the results of the calculations are reflected in the values that the transaction writes to the database. The following is the simplified pseudocode for the previous transaction:

```
Begin_Tran T1:
     Read(X(bal))
     Read(Y(bal))
     Write(X(bal))
     Write(Y(bal))
End_Tran T1;
```

It is important to realize that each transaction contains an ordered list of operations. It is also important to realize that in a multi-user transaction processing environment, there could be many transactions running in the system at the same time and their operations might be interleaved. Therefore, it is important to identify to which transaction each operation belongs. To do this, we will use a compact (but easy to understand) notation.

In this notation, read operations are represented by an "R" and write operations are represented by a "W." The R and W are followed by a number representing the transaction to which the operation belongs. We will also use generic data item names such as "X" and "Y" and put the data item name in parentheses for each operation. Applying this convention to our fund transfer transaction, we will have the following new representation of our transaction:

```
Begin_Tran T1:
     R1(X)
     R1(Y)
     W1(X)
     W1(Y)
End_Tran T1;
```

We can represent this in an even more compact notation, by using a single formula—"T1 = R1(X), R1(Y), W1(X), W1(Y)" is a straight-line representation of our T1 transaction. In this representation, it is understood that R1(X) happens before R1(Y), R1(Y) happens before W1(X), and so on, meaning that the time increases from left to right.

Now, let's assume that Alice is a customer who owns account X and account Y. Let's also assume that account X has a balance of $1500 and account Y has a balance of $500. As far as Alice is concerned, the database is in a correct state as long as the two accounts show the balances we just indicated. Suppose Alice wants to transfer $500 from account X to account Y. When the transaction completes, the database is in correct state if both accounts have a balance of $1000. If either account shows any other balance value, it would indicate that the database is in an inconsistent state (and Alice would probably be upset). In order to ensure that the database remains correct and consistent, there are four properties (the ACID properties we mentioned earlier) that every transaction must satisfy.

- The **atomicity** property of a transaction indicates that either all of the operations of a transaction are carried out or none of them are carried out. This property is also known as the "all-or-nothing" property.
- The **consistency** property of a transaction requires a transaction to be written correctly. It is the programmer's responsibility to implement the code in the program correctly—so that the program carries out the intention of the transaction correctly. For the fund transfer transaction example, if we are given two account numbers representing the source and the target accounts, the programmer has to make sure the transaction debits the balance of the first account and credits the balance of the second account. If any other modifications are made to the balances of these accounts, the program is invalid. The program implementing the fund transfer transaction must also verify the existence of the account numbers it was given by the user. In addition to these requirements, the program implementing the transaction must also check to see if there is enough money in the source account for the transfer. Note that if Alice inputs the account numbers in the wrong order, the money will be transferred in the wrong direction, that is, the reverse order of what she intended. In this case, it was Alice's mistake—there is nothing wrong with the program implementing the transaction, and it cannot

do anything about the mistake. Therefore, the burden of enforcing transaction consistency is shared between the programmer who codes the transaction and the user who runs the transaction.

- The **isolation** property of a transaction requires that the transaction be run without interference from other transactions. Isolation guarantees that this transaction's changes to the database are not seen by any other transactions until after this transaction has committed.
- The **durability** property of a transaction requires the values that the transaction commits to the database to be persistent. This requirement simply states that database changes made by committed transactions are permanent, even when failures happen in the system. When a failure happens, there are two types of changes that can be found in the database. The first type refers to all the changes made by transactions that committed prior to the failure. The other type refers to all the changes made by transactions that did not complete prior to the failure. Since changes made by incomplete transactions do not satisfy the atomicity requirement, these incomplete transactions need to be undone after a failure. This will guarantee that the database is in a consistent state after a failure. We will address durability and handling failures in conjunction with commit protocols in detail in Chapter 8.

As an example of how failures can result in an inconsistent database, assume the system loses power at some point during the execution of the fund transfer transaction. This leads to the transaction's abnormal termination. One can imagine the case where the debit portion of the transaction has completed but the credit portion has not. In this case, the savings account balance will be $1000 but the checking account balance is still $500. This obviously indicates an inconsistent state for the database. The DBMS has to guarantee that when the power is restored, the savings account balance will be restored to $1500. Let's consider another example that demonstrates the ACID properties.

Example 5.2 To see what can go wrong when two transactions that interfere with each other run concurrently (i.e., what would happen if we did not have the isolation property), consider the following scenario. Transaction T1 is transferring $500 from account X to account Y at the same time that transaction T2 is making a $200 deposit into account Y. We can show the operations of these two transactions as follows:

```
Begin_Tran T1:
     Read (account X's balance) into X
     Read (Account Y's balance) into Y
     Calculate X = X - 500
     Calculate Y = Y + 500
     Write (account X's balance) with X
     Write (account Y's balance) with Y
End_Tran T1;

Begin_Tran T2:
     Read (account Y's balance) into Y
     Calculate Y = Y + 200
     Write (account Y's balance) with Y
End_Tran T2;
```

Assuming the same starting balances as before (account X has a balance of $1500 and account Y has a balance of $500), we can say that after these two transactions complete, the database is correct if account X′s balance is $1000 and account Y′s balance is $1200. Any other values for the balances of these accounts will leave the database in an inconsistent state. Without isolation, one possible scenario interleaves the operation of T1 and T2 as "R1(X), R1(Y), R2(Y), W1(X), W1(Y), W2(Y)." This scenario results in account X′s balance being $1000 and account Y′s balance being $700, which makes the database inconsistent. Another possible scenario interleaves the operations as "R1(X), R1(Y), R2(Y), W1(X), W2(Y), W1(Y)." This scenario also produces an inconsistent database since account X′s balance will be $1000 and account Y′s balance will be $1000. Lack of isolation is the cause of the inconsistency of the database in both of these two scenarios. In the first scenario, the value of Y that was written by T1 was lost, and in the second scenario, the value of Y that was written by T2 was lost. Similar issues can occur when the temporary values of one transaction are seen by other transactions.

In environments such as banking systems or airline reservation systems, there are a large number of transactions that run concurrently against the database. In concurrent transaction processing systems like these, the isolation property enables each transaction to perceive itself as the only transaction running in the system. The concurrency control subsystem of the DBMS achieves this perception by enforcing transaction isolation using either **locking** or **timestamping**. We will discuss how isolation maintains the consistency of the database later in this chapter.

5.1.2.1 Transaction Redefined Previously, we defined a transaction to be a collection of operations against the data items of a database without talking about any of the properties the transaction must satisfy. Now that we have introduced transaction atomicity, consistency, isolation, and durability, we need to refine the definition of a transaction. Jim Gray [Gray81] defines a transaction as follows:

> A transaction is a collection of user actions mapped to operations against the data items of the database that, if runs uninterrupted by other transactions and by system failures, transfers the database from one consistent state into another consistent state. If the transaction aborts, it does not change the state of the database.

Figure 5.2 depicts the life cycle of a transaction as Jim Gray defines it. It is important to notice that when a transaction finishes (either successfully or unsuccessfully), the database is left in a consistent state. A successful termination, indicated by a commit, creates a new consistent state for the database, while an unsuccessful termination, caused by an abort, restores the consistent state prior to the start of the transaction. What is equally important is that during the execution of the transaction the database may or may not be in a consistent state. In our fund transfer example, the state of the database after the debit has been done, but before the credit is applied, is an example of an inconsistent state. The database can never be left in this inconsistent state. In case of a failure in this state, a rollback is required to undo the incomplete work of the transaction to restore the previous consistent state.

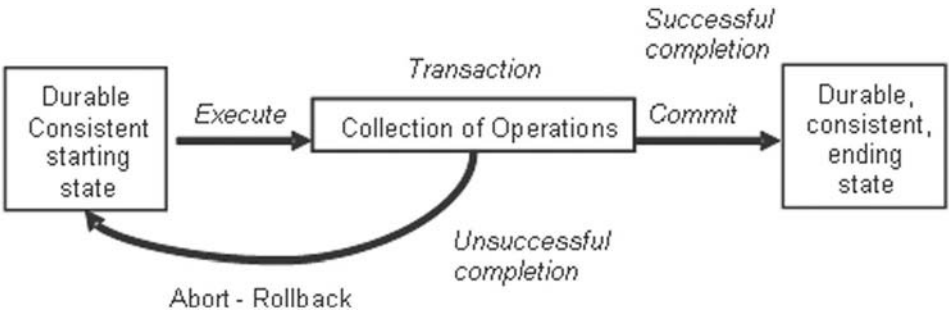


Figure 5.2 Transaction life cycle.

5.2 MULTITRANSACTION PROCESSING SYSTEMS

Our focus on concurrency control is on on-line transaction processing (OLTP) systems. An OLTP system is a system with many transactions running at any given point of time. One characteristic of these systems is the fact that transactions in an OLTP environment are short-lived transactions and make many changes to the database. These systems are somewhat different from **on-line analytical processing** (**OLAP**) systems that support **decision support system** (**DSS**) or data warehousing. Transactions in OLAP systems are long-lived and may make little changes to the data warehouse.

Banking systems, airline reservation systems, and systems that monitor the stock market are typical examples of multitransaction processing environments. At any given point in time, there may be hundreds or thousands of transactions running concurrently in the system. Some of these concurrent transactions may interfere with each other. In these cases, it is important to keep track of how the multiple transactions' operations are interleaved. This is what modern database management systems maintain as a **schedule**.

5.2.1 Schedule

A schedule is the total order of operations for a set of transactions. A schedule has sometimes been called a **history** in the industry. When different transactions run in a system, they sometimes create a **serial schedule** and other times a **parallel schedule**.

5.2.1.1 Serial Schedule A serial schedule is one that contains transactions whose operations do not overlap in time. This means that at any point in time only one transaction is running in the system. This schedule is also known as a **sequential schedule**. The schedule "S1 = R1(X), R1(Y), W1(X), R2(Y), W2(Y)" is an example of a serial schedule. This schedule indicates that transaction T1 commits before T2 starts. We show this order as "T1 $\Rightarrow$ T2," where the "$\rightarrow$" indicates commitment precedence. Figure 5.3 depicts an example of a serial schedule.

5.2.1.2 Parallel Schedule A **parallel schedule** is one that may contain transactions whose operations do overlap in time. This means that at any point in time there can be

7

REPLICATION CONTROL

Replication is a technique that only applies to distributed systems. A database is said to be replicated if the entire database or a portion of it (a table, some tables, one or more fragments, etc.) is copied and the copies are stored at different sites. The issue with having more than one copy of a database is maintaining the **mutual consistency** of the copies—ensuring that all copies have identical schema and data content. Assuming replicas are mutually consistent, replication improves availability since a transaction can read any of the copies. In addition, replication provides for more reliability, minimizes the chance of total data loss, and greatly improves disaster recovery. Although replication gives the system better read performance, it does affect the system negatively when database copies are modified. That is because an update operation, for example, must be applied to all of the copies to maintain the mutual consistency of the replicated items.

Example 7.1 This example shows how the mutual consistency of two copies of a database can be lost when the copies are subjected to schedules that are not identical. In this example, the value of data item "X" is 50 at both the LA and NY sites before transactions "T1" and "T2," as shown below, start.

```
Begin_Tran T1;
    R(X);
    X = X - 20;
    W(X);
End_Tran T1;
```

```
Begin_Tran T2;
     (R(X);
     X = X * 1.1;
     W(X);
End_Tran T2;
```

Distributed Database Management Systems by Saeed K. Rahimi and Frank S. Haug

T1 runs in LA before it moves to NY to run there. T2, on the other hand, runs in NY first and them moves to LA to run there. Based on this ordering of the transactions' operations, the following two schedules are formed at LA and NY:

```
SLA = R1(X), W1(X), R2(X), W2(X)
SNY = R2(X), W2(X), R1(X), W1(X)
```

When we apply these schedules to the copies of X in LA and NY, we end up with "X = 33" in LA and "X = 35" in NY as shown below:

```
At Los Angeles
Initial Value        X = 50
T1 Subtracts 20      X = 30
T2 Increases 10%     X = 33

At New York

Initial Value        X = 50
T2 Increases 10%     X = 55
T1 Subtracts 20      X = 35
```

Recall from Chapter 5 that SLA is a serial schedule as "T1 → T2" and SNY is also a serial schedule as "T2 → T1." Although both local schedules are serial schedules and maintain the consistency of the local copies, the system has a cycle in its global schedule, which leads to the inconsistency between the copies. To maintain the mutual consistency of the data items in the replicated database, we must enforce the rule that "two conflicting transactions commit in the same order at every site where they both run." See Section 5.2.4.2 in Chapter 5. Guaranteeing this requirement is the topic of replication control, which we will discuss in this chapter.

7.1 REPLICATION CONTROL SCENARIOS

It should be clear by now that replication control algorithms must maintain the mutual consistency of the copies of the database. One way to categorize the approaches is based on whether or not the copies are identical at all times. From this perspective, there are two approaches to replication control: **synchronous replication control** and **asynchronous replication control**.

In synchronous replication, replicas are kept in sync at all times. In this approach, a transaction can access any copy of the data item with the assurance that the data item it is accessing has the same value as all its other copies. Obviously, it is physically impossible to change the values of all copies of a data item at exactly the same time. Therefore, to provide a consistent view across all copies, while a data item copy is being changed, the replication control algorithm has to hide the values of the other copies that are out of sync with it (e.g., by locking them). In other words, no transaction will ever be able to see different values for different copies of the same data item. In asynchronous replication, as opposed to synchronous replication, the replicas are **not** kept in sync at all times. Two or more replicas of the same data item can have different

values sometimes, and any transaction can see these different values. This happens to be acceptable in some applications, such as the warehouse and point-of-sales database copies.

7.1.1 Synchronous Replication Control Approach

In this approach, all copies of the same data item must show the same value when a transaction accesses them. To ensure this, any transaction that makes one or more changes to any copy is expanded to make the same change(s) to all copies. The two-phase commit protocol (see Chapter 8) is used to ensure the atomicity of the modified transaction across the sites that host the replicas. For example, assume we have copied the "EMP(Eno, Ename, Sal)" table in all the sites in a three-site system. Also, assume transaction "T1" has the following operations in it:

```
Begin T1:
     Update EMP
     Set Sal = Sal *1.05
     Where Eno = 100;
End T1;
```

This transaction gives a 5% salary increase to employee 100. To make sure that all copies of the EMP table are updated, we would change T1 to the following:

```
Begin T1:
     Begin T11:
          Update EMP
          Set Sal = Sal *1.05
          Where Eno = 100;
     End T11;

     Begin T12:
          Update EMP
          Set Sal = Sal *1.05
          Where Eno = 100;
     End T12;

     Begin T13:
          Update EMP
          Set Sal = Sal *1.05
          Where Eno = 100;
     End T13;
End T1;
```

where T11 is a child of T1 that runs at Site 1, T12 is a child of T1 that runs at Site 2, and T13 is a child of T1 that runs at Site 3. Because of this modification, when T1 commits all three copies will have been updated accordingly. It is important to note that, during the execution of T1, employee 100's salary is locked and therefore no other transaction can see a mutually inconsistent value for this employee's salary.

The lock is removed when T1 commits, at which time T1 reveals the new salary for employee 100 in all copies of the EMP table.

It should be obvious that enforcing synchronous replication control has major performance drawbacks. Another major issue is dealing with site failures. If one of the sites storing a replica of a data item modified by T1 goes down during the execution of T1, T1 is blocked until the failed site is repaired. We will discuss the impact of site failure on transaction execution in Chapter 8. For these reasons, synchronous replication is only used when one computer is a backup of the other. In this scenario, two computers act as a hot standby for each other. Suppose we have two servers, "A" as the primary and "B" as the backup, that need to be identical at all times. In this setup, when A fails, transaction processing continues on B without interruption. Meanwhile, transactions that run on B are kept in a pending queue for application to A when it is repaired. When the A server restarts, it is not put back in service until it is synchronized with the B server. During synchronization, new transactions that arrive at either site are held in the job queue and are not run. Synchronizing A means that the pending transactions from B are applied to A in the same order that they ran on B. After all of the transactions in the pending queue have been processed against A, both servers are put back in service. At this time, transactions are processed from the front of the job queue and run against both copies.

7.1.2 Asynchronous Replication Control

In this approach, copies do not have to be kept in sync at all times. One or more copies may lag behind the others (be out of date) with respect to the transactions that have run against the copies. These copies need to eventually catch up to the others. In the industry, this process is known as **synchronization**. How and when the out-of-date copies are synchronized with the others depends on the application. There are multiple approaches to implementing the required synchronization. Most commercial DBMSs support what is known as the **primary copy** approach. This approach is also known as the **store and forward** approach. The site that is updated first is called the **primary** site, while others are known as the **secondary** sites. Some DBMS vendors call the primary site the **publisher** and the secondary sites the **subscribers**. All transactions are run against the primary site first. This site determines a serialization order for the transactions it receives and applies them in that order to preserve the consistency of the primary copy. The transactions are then queued for application to the secondary sites. Secondary sites are updated with the queued transactions using a batch-mode process. This process is known as the **rollout**. The database designer or the DBA decides on the frequency of rolling out transactions to the secondary sites.

The secondary copies can be kept in sync with the primary copy by either rolling out the transactions that are queued or rolling out a snapshot of the primary copy. To roll out queued transactions, the transactions are run from the front of queue against all secondary sites. In the snapshot rollout, the image of the primary copy is copied to all secondary sites. The advantage of rolling out a snapshot of the primary is speed. Most databases support unloading a database to a file and reloading the database from a file. Snapshot replication can be implemented using these database capabilities to speed up the synchronization of the secondary sites with the primary. Snapshot replication can be done on demand, can be scheduled, or can simply run periodically.

In asynchronous replication, the failure of the primary is troublesome, since the primary is the only copy that is updated in real time. To deal with the failure of the primary, the system may use a hot standby as a backup for the primary. When the primary fails, the standby will continue to act as the primary until the primary is repaired and is synchronized as explained in Section 7.1.1. Instead of having a fixed site as the primary, an approach that allows for a floating primary can also be utilized. In this approach, the responsibility to be the hot standby for the primary is passed from one site to another in a round-robin fashion. As another alternative, sites may also be allowed to compete to be the primary's hot standby, if the designers choose.

Other alternatives to asynchronous replication control can also be implemented. In one approach, instead of having one primary and many secondary sites, a system can have multiple primaries and a single secondary. In this case, transactions are applied to each primary when they arrive at a secondary. Since there is no immediate synchronization between the primaries, the primary copies may diverge. To synchronize all the copies, transactions that are queued at each primary are sent to the single secondary for application. This copy will then generate a serialization order that is applied to the secondary and is then rolled out to all the primaries. In another approach, as opposed to having sites designated as primary and secondary, we can have all sites act as peers. In this approach to replication, we utilize **symmetric replication** in which all copies are treated the same. Transactions are applied to the local copy of the database as they arrive at a site. This approach potentially causes the divergence of the replicas. Example 7.1 in Section 7.1.1 discusses this approach to replication control. Since the copies of the database in this approach may diverge, occasionally the system has to synchronize all the copies. Most DBMS vendors ship the necessary software to compare the contents of the copies and also provide tools for synchronization and identification of differences in the copies. When synchronization is required, tools from the DBMS allow the database designer or a DBA to synchronize the replicas before they are put back in service again.

7.2 REPLICATION CONTROL ALGORITHMS

Many replication control algorithms have been studied and proposed in the literature [Ellis77] [Gifford79] [Rahimi79] [Thomas79] [Rahimi80]. All these approaches focus on providing synchronous replication control. The algorithms can be categorized in two general categories of centralized and distributed control. In the centralized approach, the control is given to one site while in the distributed approach the control is shared among the sites. Before discussing replication control algorithms, we need to define the architectural aspects of a replicated database environment (DBE).

7.2.1 Architectural Considerations

As mentioned in Chapter 2, a database can be fully or partially replicated. A fully replicated database stores one complete copy of the database at each database server. A partially replicated database, on the other hand, does not store a copy of every table of the database at every site. Instead, some of the tables (or some of the table fragments) are copied and stored at two or more sites. To simplify the discussion of the

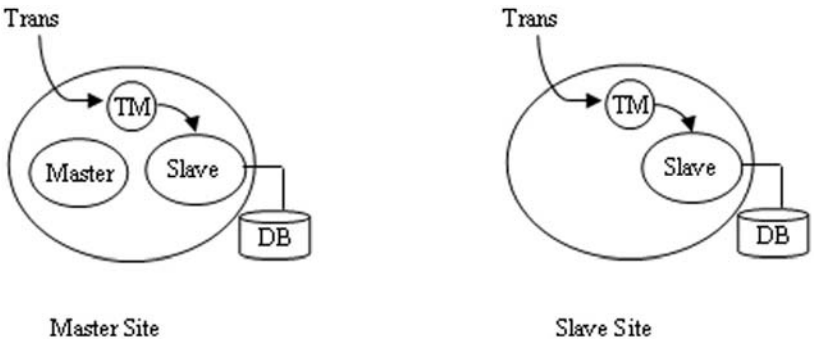


Figure 7.1 Master–salve replication control algorithm placement.

replication control algorithms, we assume the database is fully replicated. Extension of the algorithms to cover partially replicated databases is left as an exercise.

We also make the following assumptions with regard to the communication subsystem:

- Message delays are viable, meaning that there is a time delay between when a message is sent and when it is received.
- Messages are received in the same order in which they are sent. To guarantee this, the communication subsystem must use some form of queuing mechanism to order the delivery of the messages.
- Finally, we assume that all messages that are sent will be received (no messages are lost).

The last two points are referred to as "reliable messaging," which is discussed in Section 15.1.2.5. General messaging issues are also discussed in Section 14.3.1.2.

7.2.2 Master–Slave Replication Control Algorithm

The master–slave replication control algorithm was first proposed by C. Ellis [Ellis77]. In this implementation, as shown in Figure 7.1, there is one master algorithm and N slave algorithms for an N-site system. The master algorithm runs at only one site and is responsible for conflict detection. A copy of the slave algorithm runs at every site where there is a copy of the database. The implementation of this approach consists of two phases: the transaction acceptance/rejection phase and the transaction application phase.

Ellis uses a modified Petri net [Petri62] diagram notation known as an evaluation net [Nutt72]. In this notation, squares represent events, circles represent states, large arrows represent tasks, and small arrows represent transitions. A dot next to the large arrow indicates a single message being sent as one of the actions in the task, while three dots (an ellipsis) indicate a broadcast. We have used a Petri net to discuss the master algorithm in Figure 7.2. We will not show the slave algorithm diagrammatically here—this is left as an exercise for the reader. We will explain how the two algorithms work to ensure consistency of the copies of the database in the next section.

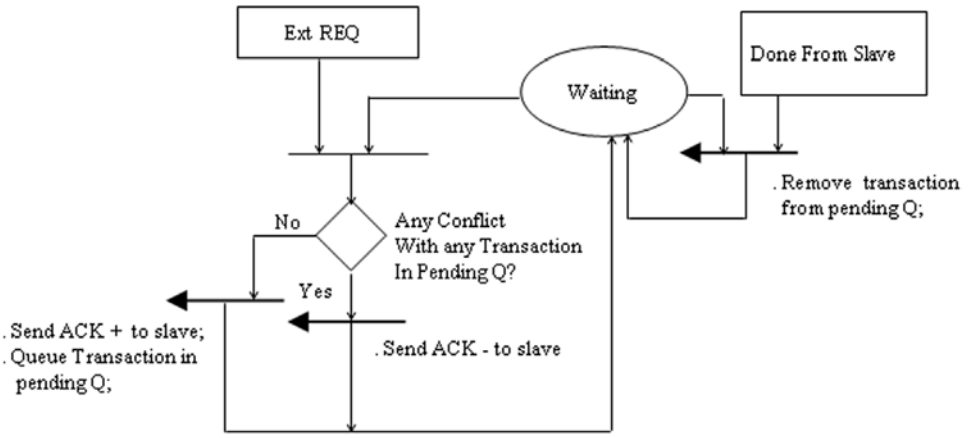


Figure 7.2 Petri net for the master algorithm in a master–slave replication algorithm.

7.2.2.1 Centralized Transaction Acceptance/Rejection Phase When a transaction enters a site, the local transaction monitor (TM) sends a request to the slave at that site asking it to run the transaction. Upon receiving this request, the slave sends a request to the master asking if it is okay to run this transaction. The box labeled "Ext REQ" in Figure 7.2 indicates this request. To be able to detect conflicts between running transactions, the master maintains a pending queue of such transactions. The master's pending queue holds all the currently running transactions that have been approved by the master (the master voted OK to each of them) but have not been committed yet. When the master receives a request from a slave, it checks for conflicts between this transaction request and all the transactions that are already in its queue. If the new request is not in conflict with any transaction in its pending queue, the master enters the transaction in its pending queue and sends an "ACK+" (positive acknowledgment) message to the requesting slave indicating that it is OK to run the transaction. If, on the other hand, the new request is in conflict with one or more of the transactions in the pending queue, the master responds with an "ACK−" (negative acknowledgment) message to the slave indicating that it is not OK to run this transaction. When the requesting slave receives a favorable response from the master, it starts the transaction application phase. On the other hand, if the response from the master is not favorable, then the slave rejects the transaction.

7.2.2.2 Centralized Transaction Application Phase When a slave receives the OK for its transaction from the master, it broadcasts a request to update by sending a "UPD" message to all other slaves. Slaves that receive this request must run the transaction and acknowledge it by sending an "ACK" message back to the requesting slave. The requesting slave waits until it receives acknowledgments from all its peers. This is an indication that the transaction has been applied successfully to all database copies at all sites. Only then does this slave send a "DONE" message to the master and the TM, letting them know that the transaction has been applied successfully. Upon receiving the "DONE" message from the slave, the master removes the transaction from its pending queue.

Like any other centralized control algorithm, the centralized voting algorithm has the drawback of overloading the center of control, which is the weakest point of the architecture. The failure of the center causes the overall system to collapse. To overcome issues of load balancing and improve resiliency, Ellis also proposed a distributed voting algorithm, which we discuss next.

7.2.3 Distributed Voting Algorithm

This implementation of the distributed voting algorithm is similar to the centralized version, in that a transaction has to be "accepted" before it is actually applied to all copies. The difference lies in how the transaction acceptance is carried out. In this approach, since there is no master algorithm, all the sites act as peers. In this scenario, they all have to "OK a transaction" before it runs. Any objection from any site will cause the rejection of the transaction. In other words, this implementation requires consensus on applying a transaction, that is, the decision to approve the transaction must be unanimous. Once consensus is reached, the transaction is applied by all the slaves (peers) at all the sites.

7.2.3.1 Distributed Transaction Acceptance Phase As in the centralized case, when a transaction enters a given site, the local TM sends a local request to the slave at that site asking it to run the transaction. Upon receiving this request, the slave broadcasts a request to all other slaves asking them if it is OK to run this transaction. The requesting slave then waits until it receives responses from all other slaves. If they all OK the transaction, the requesting slave proceeds to the transaction application phase. If, on the other hand, one or more slaves are NOT OK to run the transaction, the requesting slave rejects the transaction. Each and every site uses a set of priority-based voting rules to resolve conflicts. There are three basic conflict cases that a slave needs to handle.

Assuming sites are voting on transaction Ti, the following rules are applied:

- *Case 1:* The voting slave is not currently working on a transaction that has been initiated at its site. In this case, the slave votes OK to applying Ti.
- *Case 2:* The voting slave is currently working on transaction Tj that has been initiated at its site and Tj is NOT in conflict with Ti. In this case, the slave votes OK to Ti since both Ti and Tj can run concurrently.
- *Case 3:* The voting slave is currently working on transaction Tj that has been initiated at its site and Tj is in conflict with Ti. Since Ti and Tj cannot run concurrently, one of them has to be rejected (aborted). To sort out the conflict, age is used as a priority. If Tj has a higher priority than Ti, because it is older, the slave votes NOT OK to Ti, causing Ti to die. If, on the other hand, Tj has lower priority than Ti, because it is younger, then the slave votes OK to Ti, causing its own transaction Tj to die.

7.2.3.2 Distributed Transaction Application Phase The slave that has started the transaction acceptance phase for its transaction can start the transaction application phase when it receives an OK from all other slaves. In this phase, the slave applies the transaction to all copies by sending a new broadcast. The broadcast requests all other slaves to run the transaction. Slaves that receive this request simply run the transaction

8

FAILURE AND COMMIT PROTOCOLS

Any database management system needs to be able to deal with failures. There are many types of failures that a DBMS must handle. Before discussing failure types and commit protocols, we need to lay the foundation by defining the terms we will use in the rest of this chapter.

8.1 TERMINOLOGY

We define the terms we need here to make sure that the reader's understanding of these terms is the same as what we have in mind.

8.1.1 Soft Failure

A **soft failure**, which is also called a **system crash**, is a type of failure that only causes the loss of data in nonpersistent storage. A soft failure does not cause loss of data in persistent storage or disks. Soft failures can range from the operating system misbehaving, to the DBMS bugs, transaction issues, or any other supporting software issues. We categorize these under the soft failure, since we assume that the medium (the disk) stays intact for these types of failure. A soft failure can also be caused by the loss of power to the computer. In this case, whatever information is stored in the volatile storage of the computer, such as main memory, buffers, or registers, is lost. Again, the assumption we make is that power loss does not cause disk or database loss. This assumption does not mean that we do not deal with disk failure issues but that we treat the disk loss under the hard failure that we discuss next. Note that soft failures can leave the data stored in persistent storage in a state that needs to be addressed; for example, if the software was in the middle of writing information to the disk, and only

Distributed Database Management Systems by Saeed K. Rahimi and Frank S. Haug

some data items were written before the failure, then, obviously, the data content might be inconsistent, incomplete, or possibly corrupted. However, the persistent storage did not lose any data. This is true in the sense that the persistent storage still contains everything we wrote to it—it just does not contain all the things we intended to write to it.

8.1.2 Hard Failure

A **hard failure** is a failure that causes the loss of data on nonvolatile storage or the disk. A disk failure caused by a hard failure destroys the information stored on the disk (i.e., the database). A hard failure can be caused by power loss, media faults, IO errors, or corruption of information on the disk. In addition to these two types of failures, in a distributed system network failures can cause serious issues for a distributed DBMS. Network failures can be caused by communication link failure, network congestion, information corruption during transfer, site failures, and network partitioning. There have been many studies on the percentage, frequency, and causes of soft and hard failures in a computer system over the years [Mourad85] [Gray81] [Gray87]. We will not repeat these finding here.

As a rule of thumb, the following can be used as the frequency of different types of failures:

- Transaction failures happen frequently—maybe as many as a few times a minute. This is usually for high-volume transaction processing environments like banking and airline reservation systems. The recovery is usually fast and is measured in a fraction of a second.
- System failures (power failure) can happen multiple times a week. The time it takes to recover is usually minutes.
- Disk failures can happen once to maybe twice a year. Recovery is usually short (a few hours), if there is a new, formatted, and ready-to-use disk on reserve. Otherwise, duration includes the time it takes to get a purchase order, buy the disk, and prepare it, which is much longer (could be a number of days to a week or two).
- Communication link failures are usually intermittent and can happen frequently. This includes the communication link going down or the link being congested. Recovery depends on the nature of the failure. For congestion, typically the system state changes over time. For link failure, the link will be bypassed by the routing protocols until after the link is repaired. Sometimes the link failure is caused by the failure of a hub or a router. In this case, the links that are serviced by the hub or the router are disconnected from the rest of the network for the duration of the time the device is being repaired or replaced.

8.1.3 Commit Protocols

A DBMS, whether it is centralized or distributed, needs to be able to provide for atomicity and durability of transactions even when failures are present. A DBMS uses **commit protocols** to deal with issues that failures raise during the execution of transactions. The main issue that the commit protocols have to deal with is the ability

to guarantee the "all or nothing" property of transactions. If a failure happens during the execution of a transaction, chances are that not all of the changes of the transaction have been committed to the database. This leaves the database in an inconsistent state as discussed in Chapter 5. Commit protocols prevent this from happening either by continuing the transaction to its completion (roll forward or redo) or by removing whatever changes it has made to the database (rollback or undo). The commit protocols guarantee that after a transaction successfully commits, all of its modifications are written to the database and made available to other transactions. Commit protocols also guarantee that all the incomplete changes made by the incomplete transactions are removed from the database by rollback when a failure happens.

8.1.3.1 Commit Point A **commit point** is a point in time when a decision is made to either commit all the changes of a transaction or abort the transaction. The commit point of a transaction is a consistent point for the database. At this point, all other transactions can see a consistent state for the database. The commit point is also a restart point for the transaction. This means that the transaction can be safely undone. Finally, the commit point is a release point for the resources that the transaction has locked.

8.1.3.2 Transaction Rollback (Undo) Transaction rollback or undo is the process of undoing the changes that a transaction has made to the database. Rollback is applied mostly as the result of a soft failure (see Section 8.1.1). Rollback is also used as a necessary part of transaction semantics. For example, in the fund transfer transaction in a banking system, there are three places from which a transaction should abort. It is necessary to abort this transaction if the first account number is wrong (it does not exist), if there is not enough money in the first account for the transfer, and finally, if the second account number is wrong (it does not exist). The following shows the fund transfer transaction that is written as a nested transaction consisting of two subtransactions—a debit and a credit transaction. This nesting of the two transactions allows the program to run on a distributed DBMS, where accounts are stored on two different servers.

```
Begin Transaction Fund_Transfer(from, to, amount);
     Begin Transaction Debit
          Read bal(from);
          If account not found then
          Begin
               Print "account not found";
               Abort Debit;
               Exit;
          End if;
          bal(from) = bal(from) – amount;
          if bal(from) < 0 then
          Begin
               Print "insufficient funds";
               Abort Debit;
               Exit;
          End if;
```

```
          write bal(from);
          Commit Debit;
     End Debit;
     Begin Transaction Credit
          Read bal(to);
          If account not found then
          Begin
               Print "account not found";
               Abort Credit;
               Exit;
          End if;
          bal(to) = bal(to) + amount;
          write bal(to);
          Commit Credit;
     End Credit;
     Commit Fund_Transfer;
End Fund_Transfer;
```

This transaction transfers the "amount" from the "from" account to the "to" account. The debit subtransaction checks the validity of the account from which funds are to be taken. If that account does not exist, then the transaction is aborted. If the account exists but there is not enough money to transfer—the resulting balance after the debit is less than zero—the transaction has to abort again. The credit subtransaction does not need to worry about the amount of money in the "to" account since money will be added but needs to make sure that the "to" account exists. If the "to" account does not exist, then the transaction aborts again.

If all three abort conditions are false, then the transaction is at the commit point. At this point, as mentioned before, the user needs to make a decision to commit the transaction or roll it back. For an ATM transaction, this point in time is when the ATM machine gives the customer the option to complete the transaction or cancel it. At this point, each account's balance has been updated and everything is ready to be committed. The two database servers—local systems where the accounts are stored—have the accounts' balances still locked. Once the user decides to commit the transaction, the locks are released and the new accounts' balances are available to other transactions.

8.1.3.3 Transaction Roll Forward (Redo) Transaction roll forward or redo is the process of reapplying the changes of a transaction to the database. Since the changes are reapplied to the database, typically a transaction redo is applied to a copy of the database created before the transaction started. As we will discuss in Section 8.2, redo is mostly necessary for recovery from a hard failure.

8.1.4 Transaction States

A transaction goes through the following set of steps during execution:

```
1. Start_Transaction
2. Repeat
```

```
     2.1 Read a data item's value
     2.2 Compute new values for the data item
     2.3 If abort condition exists then abort and exit
     2.4 Write the new value for the data item
     2.5 If abort condition exists then abort and exit
3. Until no more data items to process
4. Commit
5. End_Transaction
```

Statement 1 indicates a transaction's activation by a user or the system. Similarly, statement 5 indicates the successful termination of the transaction. The transaction can also terminate unsuccessfully, as shown in statements 2.3 and 2.5, if abort conditions exist. Some DBMSs require explicit "Start Transaction" and "End Transaction" while others use implicit start and end statements. For example, in Oracle a begin transaction is assumed when changes are made to the database. In this context, an implicit "End Transaction" is assumed when a commit is issued. On the other hand, in SQL Server the "Start Transaction" and "End Transaction" are explicit. As seen in the code snippet above, once a transaction starts, it repeatedly reads the data items from the database, calculates new values for them, and writes new values for the data items to the database. Obviously, in this algorithm we assume the operation being performed inside the transaction is an update operation. In case of an insert into the database, no data items need to be read. Conversely, when a delete command is executed, no new values are written to the database.

One formal specification of a transaction uses a finite state automaton (FSA) that consists of a collection of states and transitions. This is typically shown as a state transition diagram (STD). In a STD, states are shown as circles and transitions as arrows, where the tail is connected to the state the program leaves and the tip is connected to the state the program enters. The state of the STD that a transaction is in at the time of a failure, tells the local recovery manager (LRM) what needs to be done for a transaction. Figure 8.1 depicts the STD for a transaction.

In this diagram, the double-lined circles are terminal states and the single-lined states are transitional. The "Idle" state corresponds to when the transaction is not running.

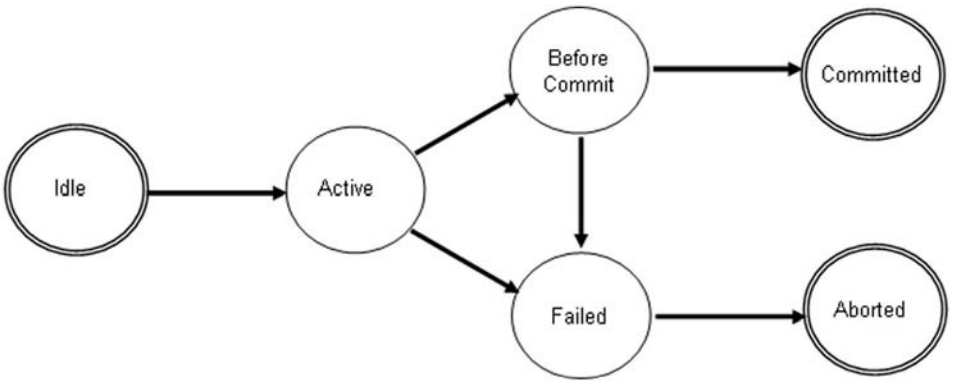


Figure 8.1 State transition diagram for a transaction.

Once started, a transaction changes state to the "Active" state. In the "Active" state, the transaction performs its work. In this state, the transaction reads, performs calculations, and writes data items. If any of the abort conditions hold, the transaction issues an abort and changes state to the "Failed" state. Entry into the "Failed" state causes the local recovery manager to undo the transaction. The transaction enters its commit point by transitioning to the "Before Commit" state. As mentioned before, at the commit point a transaction holds a consistent state for the database. It can back out of the transaction by canceling the transaction or it can complete the work by committing. Cancellation is performed by a transition from the "Before Commit" state to the "Failed" state, which causes the LRM to undo the transaction. The successful termination is initiated by transitioning from the "Before Commit" state to the "Committed" state.

It is important to remember the following:

- In the "Active" state, the transaction has not completed all the work necessary—not all changes have been made.
- In the "Before Commit" state, the transaction has completed all the work necessary—all transaction changes have been made.
- In the "Failed" state, the transaction has decided to abort or it has been forced by the DBMS to abort due to concurrency control and/or deadlock issues.
- In the "Committed" state, the transaction has terminated successfully.
- In the "Aborted" state, the transaction has terminated unsuccessfully.

A careful reader notices that actions of a transaction when it commits or aborts depend on whether or not the transaction changes have been written to the database. In other words, the update mode of the DBMS is important in what needs to be done to commit or rollback a transaction.

8.1.5 Database Update Modes

A DBMS can use two update modes:

- Immediate update mode—a DBMS is using immediate updates, if it writes the new values to the database as soon as they are produced by a transaction.
- Deferred update mode—a DBMS is using deferred update, if it writes the new values to the database when a transaction is ready to commit.

In the case of the immediate update, the old value of a data item being changed is overwritten in the database by the transaction. As a direct result of this, it is not possible to rollback the transaction if the old value of the data item is not stored anywhere else. Keep in mind that maintaining the old value of the data item in memory is not sufficient since the contents of memory will be lost in the case of power loss or system failure. Therefore, we need a safe place where the old values of data items can be stored. A transaction log is used for this purpose.

8.1.6 Transaction Log

A **transaction log** is a sequential file that stores the values of the data items being worked by transactions. Like the blocks written on a magnetic tape, a transaction log is

a sequential device. To reach a certain record within the log, the log must be processed sequentially, either from the beginning or from the end. Typically, log records for a given transaction are linked together for ease of processing.

The transaction log is used for two purposes. The first use of the log is to support commit protocols to commit or abort running transactions. The second use of the log is to allow recovery of the database in case of a failure. In the first case, we use the log information to redo—repeat the changes of a transaction—or undo—remove the changes of a transaction from a database. Redo and undo are sometimes called transaction roll forward and rollback, respectively. When a transaction is rolled forward its changes are redone—rewritten to the database. When a transaction is rolled back, its changes are undone—removed from the database. In the second case, the log information is used to recover a database after a power failure or a disk crash. As we will discuss later, if power fails, incomplete transactions have to be undone. The log information is used to achieve this. Also, as we will discuss later, if the disk crashes, completed transactions need to be redone. Again, the log information is used to achieve this.

Information that is written to the log is used for recovering the contents of the database to a consistent state after a transaction rollback or after a failure. Without this information, recovery may not be possible. Technologically, today the safest place a log can be stored is on the disk. However, if the disk fails and the log is on it, the log is lost. That is, of course, not too troubling if the database is still intact. Losing the log and the database at the same time, obviously, is disastrous.

To make sure this is not the case, one of the following alternatives can be utilized:

1. Use tapes for the log. Although this separates the database storage (disk) and log storage (tape), this is not typically the choice of DBAs because writing to and reading from the tape is very slow. Some of the DBMSs, such as Oracle, archive the inactive portion of the log—the log portion that pertains to transactions that have been competed—on tape.
2. In smaller systems where the log and the database share the same disk, one can use different disk partitions for the database and the log files. Doing so reduces the probability of loss of the database and the log at the same time when a portion of the disk is damaged. In this case, if the disk completely fails or the disk controller goes bad, both the log and the database will be lost.

To guard against the issue with the second alternative, we can store the log and the database on separate disks. This is typically the case for larger systems where the database may require more than one disk for its storage. In this case, we can use a separate disk for the log files. In very large systems, a **RAID** (**redundant array of independent disks**) system may be used to provide for recovery of the disk contents by rebuilding the contents of a failed disk from other disks. Regardless of the alternative used for maintaining the log, the goal is to keep the log safe from power loss and/or disk failure. The storage type that is used for the log is known as the "stable storage" as categorized below.

8.1.7 DBMS Storage Types

There are three types of storage in the storage hierarchy of a DBMS:

1. *Volatile Storage (Memory).* Memory is a fast magnetic device that can store information for the computer to use. It is called volatile storage since loss of

electrical power causes it to lose its contents. That is why memory is used for temporary storage of information to serve the need of the CPU. During normal operations of the system, memory-based log files can be used for database rollback, since during the rollback, the transactions are still running and the memory contents are still intact. On the other hand, the memory cannot serve as a storage medium for logs required for database recovery after a failure since the memory contents of a failed system are lost.

2. *Nonvolatile Storage (Disk, Tape).* Disk and tape are magnetic devices that can store information in a way that withstands shutdowns and power losses. The information written to a disk or a tape stays intact even when the power is disconnected. Although disks and tapes are nonvolatile with respect to power loss, they are still volatile to medium failure—a disk crash causes the loss of information on the disk. As a result, a log that is stored on the disk is lost when the disk crashes.
3. *Stable Storage.* Stable storage is a storage type that can withstand power loss and medium failure. The log is written to stable storage so that the information in the log is not lost when the system is subjected to power failure or disk failure. DBMSs usually maintain duplicate or triplicate copies of the log files for resiliency. Lampson and Sturgis assume that replicating the log on the disk (persistent storage) is stable [Lampson76]. For the rest of our discussion in this chapter, we also assume a replicated log on the disk and we refer to the disk as stable storage.

8.1.8 Log Contents

As mentioned earlier in this chapter, the log is a sequential device consisting of a set of records. A log record stores information about a change from a single transaction. In addition to the time of the change, specific information about the transaction ID, the DML operation, the data item involved, and the type of operation are recorded. Specific record contents depend on the mode of update. To represent a log record, we use the notation "<record_type>." A typical transaction log has records of the type <transaction start>, <transaction commit>, <transaction abort>, <insert>, <delete>, <update>, and <checkpoint>. Obviously, since read operations do not change data item values, they are not tracked in the log. There are many types of log records that support different strategies for updating the database—immediate versus deferred—and recovery alternatives for power and disk failures. We will start by investigating the transaction update modes and their impact on what the log should contain.

8.1.8.1 Deferred Update Mode Log Records In this mode of update, the DBMS does not write the changes of a transaction to the database until a transaction decides to commit (the commit point). There are two basic approaches used for deferred updates. Özsu and Valduriez [Özsu99] outline the details of these approaches in what they call "out-of-place" update. We briefly mention the two approaches here.

The first approach uses the concept of differential files, which has been used in the operating system for file maintenance for many years. Severance and Lohman proposed to extend the concept to database updates [Severance76]. In this approach, changes are

not made to the database directly, but instead, they are accumulated in a separate file as the difference. A file could accumulate changes from a single transaction or from multiple transactions. There are two approaches to merging the changes from the differential files with the database. A transaction's changes can be merged into the database when the transaction commits. Alternatively, changes can be left in the file and be used as the new values for the changed data items. In this case, the DBMS has to keep track of the new values for data items that have been worked on in the files. Once these files become too large and the overhead becomes too high, the files are merged into the database and the differential files are discarded.

The second approach uses the concept of "shadow paging" [Astrahan76]. In this approach, the database page containing the data item being changed is not modified. Instead, the change is made to a copy of the page called the "shadow page." The actual database page holds the old data item value (used for undo) and the shadow page contains the new data item value (used for redo). If the transaction decides to commit, the shadow page becomes part of the database and the old page is discarded. Logging in conjunction with shadow paging has been used to enable database recovery after a failure in IBM's System R [Gray81].

As a direct result of deferred updates, for each data item being changed, the old value (also known as the "before-image") is in the database until just before the transaction commits. Therefore, the log only needs to tack the new value (also known as the "after-image") for the data item being changed in the log. Therefore, in the deferred update mode, the old values of the data items being changed by a transaction do not need to be written to the log. To see what is stored in the log for a database that uses the deferred update mode, let T5 be a fund transfer transaction that transfers $200 from account A1 with the balance of $1000 to account A2 with the balance of $500. For this example, the records written to the log for T5 are

```
<T5, t1, Start>
<T5, t2, update, Account.A1.bal, after-image = $800>
<T5, t3, update, Account.A2.bal, after-image = $700>
<T5, t4, Before Commit>
<T5, t5, Commit>
```

Notice that every record has a time marker indicating the time of the event. The first record indicates that transaction T5 has started. The second and third records track the writes that T5 makes, the time of these events, and the after-images for the data items being changed. The fourth record indicates that there will be no more changes made by this transaction; that is, the transaction has entered its commit point at time t4. This log shows a successful completion of T5, which is indicated by the existence of the commit record at time t5. During normal operations, transferring the new values for the two data items from the memory buffers is followed by writing a commit record to the log. If, on the other hand, T5 had decided to abort the transaction, the log would have contained the following records:

```
<T5, t1, Start>
<T5, t2, update, Account.A1.bal, after-image = $800>
<T5, t3, update, Account.A2.bal, after-image = $700>
```

```
<T5, t4, Before Commit>
<T5, t5, Abort>
```

For deferred updates, when T5 aborts there is no need to undo anything. That is because T5 does not write anything to the database until it decides to commit. If a failure happens, depending on the time and type of failure, T5 needs to be undone or redone (see Section 8.4). If T5 needs to be redone, the after-images from the log are used. Again, there is nothing to do to undo T5.

8.1.8.2 Immediate Update Mode Log Records In this mode of update, the DBMS writes the changes of a transaction to the database as soon as the values are prepared by the transaction. As a direct result of this, the old values of the data items are overwritten in the database. Therefore, the log not only needs to track the after-images but also the before-images of the data items changed by the transaction. If T5, as discussed above, runs a system with immediate update, the log records will look like the following:

```
<T5, t1, Start>
<T5, t2, update, Account.A1.bal, before-image = $1000,
                 Account.A1.bal, after-image = $800>
<T5, t3, update, Account.A2.bal, before-image = $500,
                 Account.A2.bal, after-image = $700>
<T5, t4, Before Commit>
<T5, t5, Commit>
```

When T5 decides to commit, the new values for the two data items are already in the database. Therefore, the only action required is indicating the successful completion of the transaction by writing a "Commit" record to the log. In immediate update mode, the DBMS has access to the before-and after-images of data items being changed in the log. The DBMS can undo T5 by transferring the before-images from the log to the database and can redo T5 by transferring the after-images of the data items to the database when needed. It should be obvious that in a multitransaction system the records that different transactions write to the log are interleaved.

The log record contents discussed above are generic and do not indicate implementation by any specific DBMS. Most DBMS vendors use two or three additional fields in each record to help traverse the records for a given transaction. A linked list is used to connect together all records that belong to the same transaction. Table 8.1 depicts an example of log records for four transactions "T1," "T2," "T3," and "T4," including the link fields (Record ID and Link Info in the table). This information is helpful in locating records for a given transaction quickly during recovery from a failure. Another point worth mentioning is that commercial DBMSs do not include "before commit" and "failed" records in the log and therefore eliminate these states. These states are recorded as part of the commit and the abort states. In other words, the state transition diagram for transactions is reduced to what is shown in Figure 8.2.

This approach speeds up the log management for committed and/or aborted transactions. The assumption is that most of the time transactions succeed and they are not subjected to failures. As a result, it seems unnecessary to do a lot of recordkeeping for dealing with failures. The drawback to this approach, therefore, is the payback when failures do happen.

TABLE 8.1 Example of Log Records

Record ID	Tid	Time	Operation	Data Item	Before-Image	After-Image	Link Info
1	T1	t1	Start				3
2	T2	t2	Start				2
3	T1	t3	Update	Account.A1.bal	1000	800	8
4	T2	t4	Insert	Account.A3.No		1111	5
5	T2	t5	Insert	Account.A3.bal		750	6
6	T2	t6	Update	Account.A4.bal	600	550	9
7	T3	t7	Start				17
8	T1	t8	Before Commit				11
9	T2	t9	Before Commit				10
10	T2	t10	Commit				2
11	T1	t11	Commit				1
12	T4	t12	Start				13
13	T4	t13	Delete	Account.A5.No	2222		14
14	T4	t14	Delete	Account.A5.bal	650		15
15	T4	t15	Failed				16
16	T4	t16	Abort				12
17	T3	t17	Update	Account.A1.bal	800	1200	18
18	T3	t18	Before Commit				19
19	T3	t19	Commit				7

As an example, assume a system that does not write the "Before Commit" to the log to indicate that the transaction has finished preparing the new values for the data items. When a failure during the commitment of the transaction happens, since there is no "Before Commit" record in the log, the LRM has to assume that the transaction was active when the failure happened and will have to rollback the transaction. It should be easy to see that the existence of the "Before Commit" record in the log tells the LRM to commit the transaction using the new values of the changed data items from the log. We encourage readers to map the structure discussed above with specific DBMS

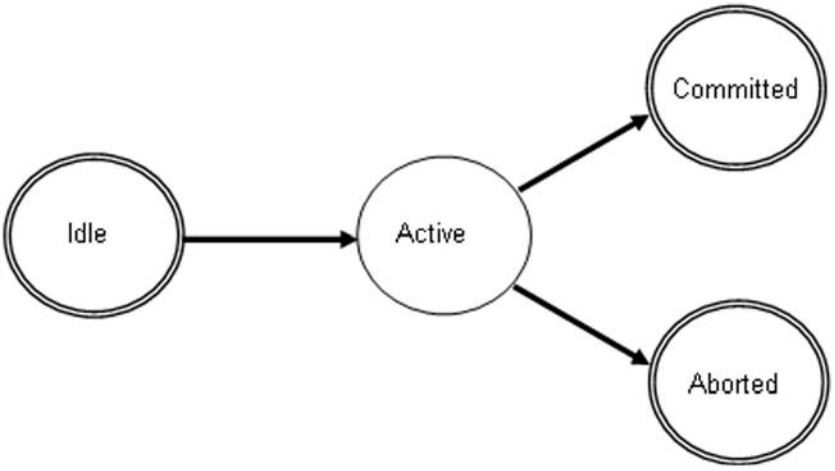


Figure 8.2 Reduced state transition diagram for a transaction.

implementation. For example, the following displays an update statement that changes the balance of account 1111 and the log records that are extracted by Oracle Log Miner from the log running the select statement shown against the "v$logmnr_contents" dictionary view.

```
Update account
Set bal = 800
Where No = 1111;
Commit;

SELECT  sql_redo, sql_undo, ph1_name, ph1_redo, ph1_undo
FROM    v$logmnr_contents;

update ACCOUNT set BAL = 800 where ROWID = 'AAABGpAABAAABdQAAA';
update ACCOUNT set BAL = 700 where ROWID = 'AAABGpAABAAABdQAAA';
BAL 800 700
```

As can be seen from the results of the select statement, Oracle records the actual SQL statement in the log record to be able to undo or redo a column change. In addition, Oracle tracks the operation type, before-image, and after-image for the column being changed. Note that there are many other columns used in this dictionary view in Oracle that we did not print. These columns record transaction ID, transaction name, timestamp, table, tablespace, and so on.

8.2 UNDO/REDO AND DATABASE RECOVERY

Now that we have discussed what the log for immediate and deferred update modes contains, we can discuss what actions are required to recover a database when a failure happens. There are only two actions the DBMS recovery manager performs to recover a database. These actions are the redoing or undoing of transactions. The question that needs to be answered is, "What action is required for each type of recovery?" The answer depends not only on the mode that the system uses for updating the database but also on the type of failure and on the state that each transaction is in when the failure happens.

8.2.1 Local Recovery Management

A DBMS tracks changes that transactions make to the database in the log for transaction abort and recovery purposes. To be able to do this, the DBMS utilizes the local recovery manager (LRM) to collect the necessary logging information. As the local transaction monitor (LTM) runs through the transaction steps, it passes the necessary information to the local scheduler (LS) for processing (see Chapter 5 on transaction processing). Each LS subsequently passes the request to the LRM. Most DBMSs today do not read or write information using only a single table row. Instead, the unit of transfer between the memory and the database is a larger unit of storage. A DBMS uses the concept

of a **block** or a **page** as the smallest unit of storage for the database. The page size in some databases, like SQL Server, is fixed at 8 Kbytes where each Kbyte is 1024 bytes. In some other systems such as Oracle, the page size can be decided by the DBA and can be 4, 8, 16, or 32 Kbytes. To read a row in a table, a DBMS needs to transfer the page that contains the desired row into memory. Most DBMSs also do not simply bring a single page into memory. Instead, they transfer the **extent** that contains the page. An extent is typically eight contiguous pages, which is the smallest allocation unit in Oracle and SQL Server today. For simplicity, for the rest of the discussion in this chapter, we assume that a single page is the unit of transfer.

Storage of information in the main memory has to match the storage of information on the disk. A DBMS assumes the memory is divided into a set of equal size blocks called **frames**. A frame can hold exactly one page. Any page can go into any frame available. To speed up the reading and writing of information, a DBMS sets aside a number of main memory frames known as the **cache** or the **buffer**. The cache or buffer size is decided by the DBA and is measured in the number of frames or pages. The program that manages the transfer of information between the buffer and the stable storage (the disk) is known as the local buffer manager (LBM). A read is complete when the page that contains the desired information is transferred from the disk into a frame in memory. If the page containing the desired information is already in the buffer, the read does not necessitate a disk access. Similarly, it is assumed that a write is complete when the changes are made to the target buffer in memory. It is the responsibility of the local recovery manager to write the page (or the extent where the page is) to the stable storage successfully. This obviously requires proper log management by the LRM that we will discuss next. Figure 8.3 shows the local recovery manager and its interfaces to the LTM, LS, and LBM.

8.2.1.1 Database Buffers and Log Buffers A DBMS maintains two types of buffers—the **database buffers** and the **log buffers**. The database buffers host database pages that have been read or written by transactions. The pages that have been written are sometimes called **dirty pages** as well. When a required page is not in the cache, it is **read** or **fetched**. When a page needs to be written to the stable storage it is **flushed** or **forced**. We would like to cache as much information as possible to speed up reading and writing.

Write-ahead-logging(WAL) [Lindsay79] was proposed to protect against loss of information being written to the database. Any change to the database is recorded in two places—in a database page in the database buffers, and in a log page in the log buffers. As long as these pages are maintained in memory, there should be no problem. If the contents of memory are lost—due to a failure—transactions whose changes were not recorded in the database must run again. According to WAL, once we decide to flush the dirty pages to the database, the log pages must be written to the stable storage before the dirty pages are written to the database on disk. That is because once we have successfully written the log in the stable storage, the database changes are guaranteed.

To see how this is possible, consider the fund transfer transaction we discussed earlier in this chapter. There are two data items that this transaction changes. Let us assume these two data items are on two separate pages in two separate extents, and that

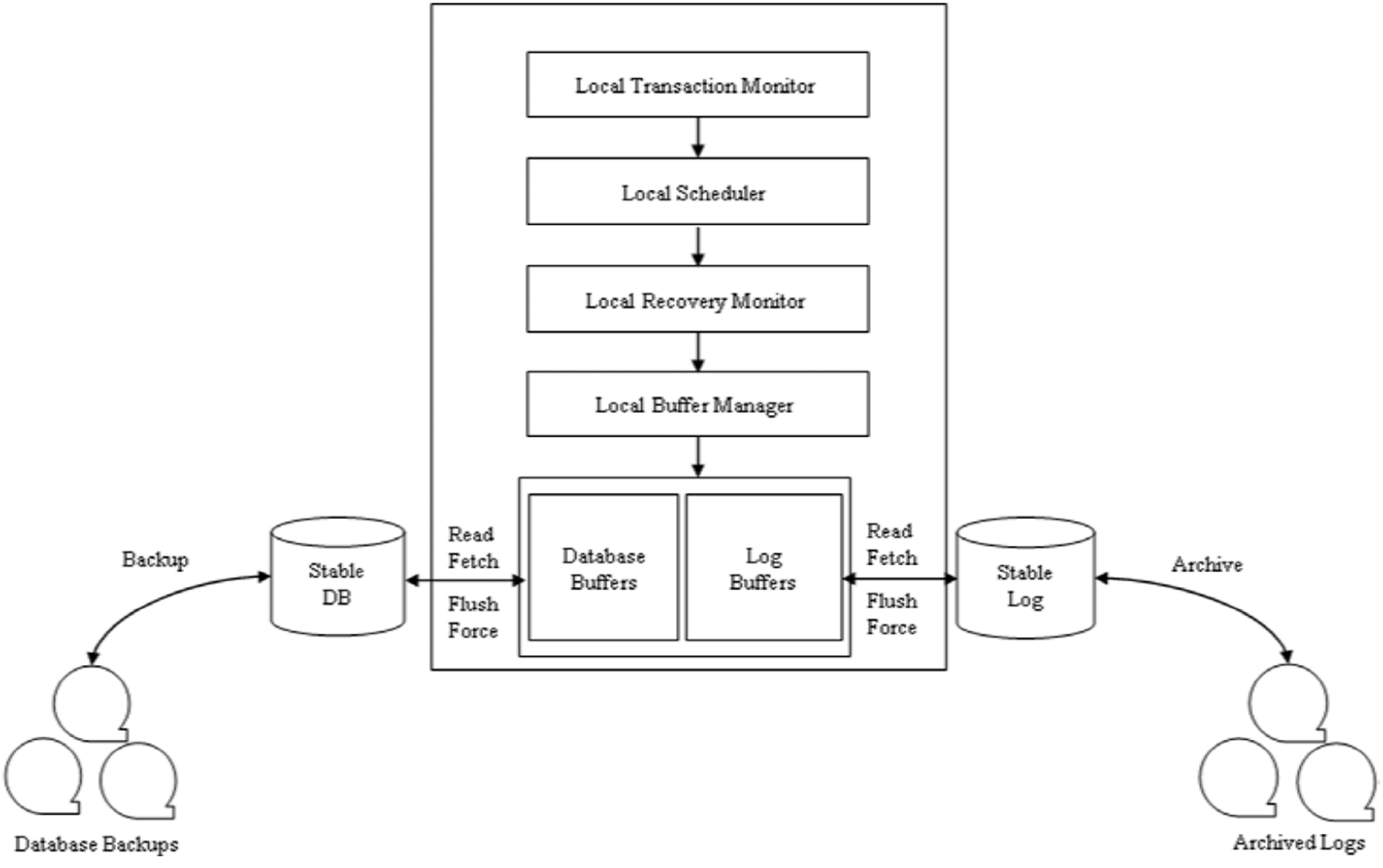


Figure 8.3 Local recovery manager's interfaces.

the changes are recorded in one log page. To commit this transaction, the LRM issues a force command to the LBM for the log page. The LBM may fail to complete this command or may succeed. In case of a failure, the contents of the memory buffers are lost. When the system restarts, the fund transfer transaction must be run again. In the absence of failures, the page is successfully written to the log in the stable storage. After the log page force command is completed, the LRM sends a force database page command to the LBM to write the two dirty pages for the transaction to the database. Again, there are two possible scenarios. In the first scenario, there are no failures, which allows the LBM to successfully write. In the second, during the writing of the dirty pages, there is a failure. In this case, after the failure is repaired, the changes can be made to the database pages by transferring the after-images from the log in the stable storage to the database pages.

8.2.1.2 Log Archival and Database Backups Log records are associated with two types of transactions—transactions that are still running (**active transactions**) and transactions that have either committed and/or aborted (**completed transactions**). It should be obvious that the log records pertaining to the running transactions need to be readily available and therefore should be cached in the memory buffers. This speeds up the rollback of currently running transactions. Once a transaction commits (or aborts), its log records are not going to be needed unless there is a failure. As a result, the log records that correspond to terminated transactions can be transferred to the disk to make memory available for active transactions.

Note that if the amount of memory we allocate to the log buffers is not large enough, there may not be enough memory for all active transactions in the cache. This can be the case for a high-volume, concurrent transaction processing environment. In these environments, the log tends to grow very rapidly. The log size could easily reach and exceed the amount of memory allocated to the cache. Therefore, it is not possible to maintain the entire log in memory due to lack of space. In this case, even some of the active transactions' log records have to be forced to the disk.

On the disk, the log typically consists of two or more log files. For example, an Oracle 10 g log consists of three log files by default. The space on these files is considered sequential from File 1 to File 2, and File 2 to File 3. As the space on File 1 is used up, File 2 becomes the active file. Once the space on File 2 is completely used, File 3 becomes the active file. Once the space on all of the log files has been used, the DBA has to either archive inactive (committed/aborted) transaction records to the tape or add more files to the log to provide space for more logging. Otherwise, the DBMS cannot continue. **Note:** Although in Figure 8.3 we refer to archiving as writing the log or the database pages to magnetic tapes, some DBAs prefer archiving of information on disks instead of tapes.

Despite the fact that the database and the log are maintained on the stable storage, recovery from a failed database is not possible without a database backup. We can back up a database on a disk during the normal operation of the system. The backup on the disk is periodically archived onto a set of tapes for recovery from a major disk failure when the database is lost. We will discuss the alternatives for the database backup in the next section.

8.2.1.3 Backup Types DBMSs use two alternatives to back up the database—a **complete backup** or an **incremental backup**. A complete backup is a backup of the

entire database contents. This is also known as a **full-image** backup. This backup can be done when the database is quiescent—there are no transactions running in the system. This type of backup is also known as **cold backup**. Alternatively, the backup can also be done when transactions are running in the system. This type of backup is known as a **hot backup**. When the backup is a cold backup, the log can be cleared since all transactions prior to the backup time have terminated. If, on the other hand, the backup is a hot backup, the log from previous cold backup is required for the recovery process. Figure 8.4 depicts examples of a cold and a hot backup.

In Figure 8.4a, the backup is a cold backup since no transactions were running at the time of the backup. In this case, since every transaction prior to the backup has terminated, there is no need to keep the log from prior to the backup. In Figure 8.4b, a hot backup was performed when transactions T2 and T3 were running. As a result, the log from prior to the last backup is needed. This also indicates the fact that as long as the backups are hot the log needs to be kept. Therefore, the log is probably going to grow very large. To prevent this from happening, a DBA must perform a cold backup periodically to be able to discard the log from the previous hot backups. Note also that a cold backup always corresponds to a consistent point of time for the database since all active transactions have been completed. Some DBMS vendors provide the capability to force the database to be quiescent. If the DBMS provides for the quiescent

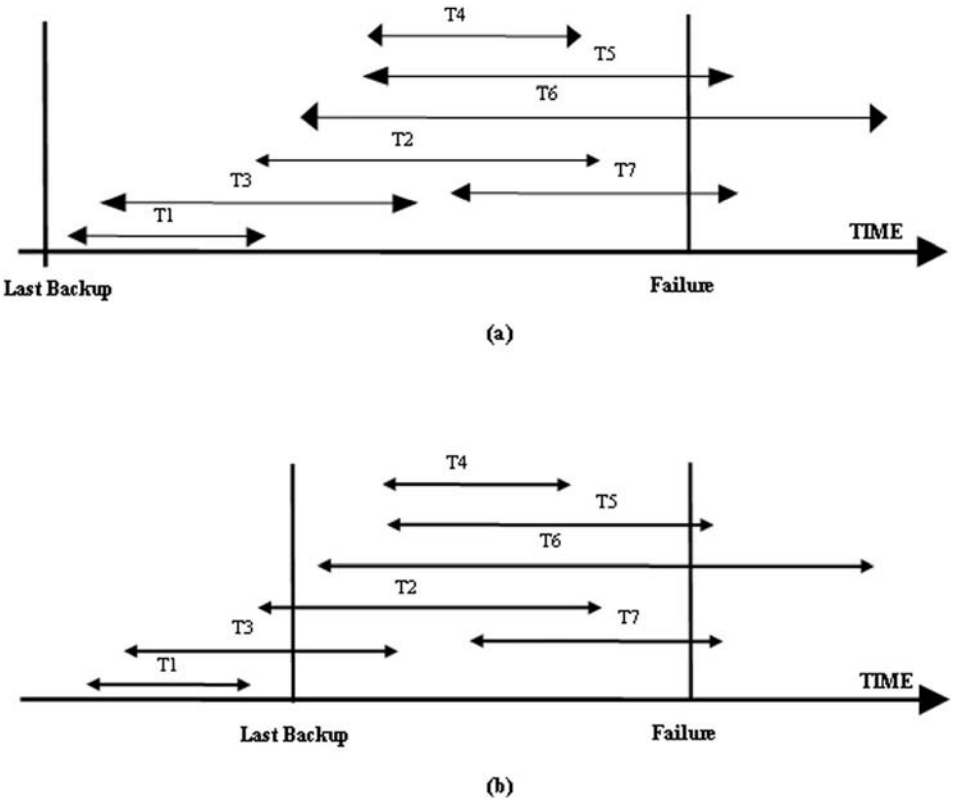


Figure 8.4 Examples of (a) cold and (b) hot backups.

option, then the DBA can force it, perform a cold backup, and discard the old log. Without an automatic quiescent option, the DBA needs to establish the point manually.

The following options can be used to force a quiescent point for a database:

- Put the database in read-only mode until after the backup.
- Bring the database off-line.
- Halt application processes.

An **incremental backup** is a backup that contains only the changes to the database from the previous complete backup or incremental backup. Since performing a complete database backup is time consuming, a DBA performs multiple incremental backups in between two consecutive complete backups. When needed, a complete image of the database is formed by merging the previous complete backup with individual incremental backups in order, up to the point of failure. Incremental backups are done when transactions are running—they are hot backups.

8.2.1.4 Rolling a Database Forward A compete backup is a consistent snapshot of the database used for recovery. A complete backup is required for recovery from a failed disk when the database is lost. However, this is not the only use of the backup. The backup is also used for other types of recovery, as we will discuss later (see Section 8.5.2). A backup is used to restore the database to a consistent point in the past (as compared to the current time). To recover a failed database, we need to reapply those transactions that have committed between the backup time and the present to the restored image of the database. The LRM achieves this by using the log to redo committed transactions after the failure. This action is known as rolling the database forward.

8.2.1.5 Rolling a Database Back The LRM rolls back a database by undoing the effects of one or more transactions. It starts with the current image of the database and rolls back transactions. This action is required when the current image of the database is inconsistent and we need to restore the database to a consistent point in the past.

8.3 TRANSACTION STATES REVISITED

Recall transaction states outlined in Section 8.1.4. We can extend the steps for a running transaction to include logging to support the transaction abort and database recovery. Note that what is logged depends on the mode of update—deferred versus immediate—as shown below (Sections 8.3.1 and 8.3.2).

8.3.1 Deferred Update Transaction Steps

The following depicts the transaction steps extended to include logging for a system that implements deferred updates.

```
1. Start_transaction
2. Log "Start"
3. Repeat
     3.1 Read a data item's "before image"
```

9

DDBE SECURITY

Bradley S. Rubin
University of St. Thomas

Security is a key facet of any database deployment. It is essential that we **authenticate** database users (ensuring that they are who they claim to be), allow only **authorized** users access to information, and maintain overall system data integrity. There are also many subtle security issues, specific to databases, such as SQL injection and inference attacks. Distributed database environments (DDBEs) require communication, so we must ensure not only that the data in databases is secure but that the communication links between users and their data and among the communicating DDBE components are also secure.

This chapter provides an overview of database and communications security. We begin by covering the basics of cryptography and outline the key building blocks and some common algorithms and best practices using those building blocks. Next, we will examine several higher-level security protocols composed of these building blocks. We then look at some security issues specific to communications and data. After examining some high-level architectural issues, we conclude this chapter with an example of how pieces in this chapter integrate.

For a comprehensive overview of security engineering, see [Anderson08]. For a more detailed look at cryptographic building blocks and specific algorithms, see [Stallings05] and [Ferguson03]. For an expanded database-specific security discussion, see [Natan05].

9.1 CRYPTOGRAPHY

Cryptography is the science of creating secrets. **Cryptanalysis** is the science of breaking secrets. The related science of hiding secrets is called **steganography**. To create a secret, we can use either **codes**, which map whole words to other words, or **ciphers**, which map individual characters (bytes) to other characters (bytes). The use of codes

Distributed Database Management Systems by Saeed K. Rahimi and Frank S. Haug

declined after World War II and today ciphers are most commonly used with digital data. The original (unencrypted) words or characters are referred to as the original message, the unencrypted message, or more specifically as the **plaintext**, while the characters we map to are referred to as the encrypted message or, more specifically, as the **ciphertext**.

Specific cryptographic functions include the following:

- **Confidentiality** keeps messages private between parties even in the face of eavesdroppers attempting to snoop while data is transported over communication networks or while it resides in the database.
- **Authentication** allows message receivers to validate the message source and to ensure the integrity of the message.
- **Nonrepudiation** validates the message source so strongly that the message sender cannot deny being the message sender (this is a stronger form of authentication).

There are a number of cryptographic building blocks that provide these functions:

- **Conventional cryptography** provides confidentiality with previously distributed keys.
- **Message digests (MDs)** and **message authentication codes (MACs)** provide authentication.
- **Public key cryptography** provides confidentiality without prior key distribution.
- **Digital signatures** provide nonrepudiation.
- **Digital certificates** and **certificate authorities** authenticate public keys.

We will now examine each of these cryptographic building blocks in more detail.

9.1.1 Conventional Cryptography

Conventional cryptography is a simple concept. Suppose we have two parties who want to communicate securely with each other; let's call them Alice and Bob. Further suppose that Alice and Bob each have their own copy of the same secret **key**, which is a random string of bits. When Bob wants to send a message to Alice and wants to ensure that no eavesdroppers can read it, he **encrypts** his message with an encryption algorithm that uses his copy of the secret key before he sends it. After receiving the encrypted message from Bob, Alice uses a corresponding decryption algorithm that uses her copy of the same secret key to **decrypt** the message so she can read it. If Alice wants to send a message to Bob, she can use exactly the same process (sending an encrypted message to Bob by encrypting her message using her copy of the secret key and then sending the encrypted message to Bob so that he can decrypt the message using his copy of the key). This is shown in Figure 9.1, with M representing the message, E representing the encrypt operation, D representing the decrypt operation, and "secret" representing the secret key.

There are a variety of algorithms that can provide the encrypt and decrypt operations, but we always assume that the algorithm is known (or at least knowable). The security of these algorithms does not depend on the obscurity of the algorithms themselves; it depends solely on the strength and obscurity of the secret key. Because a key is just a

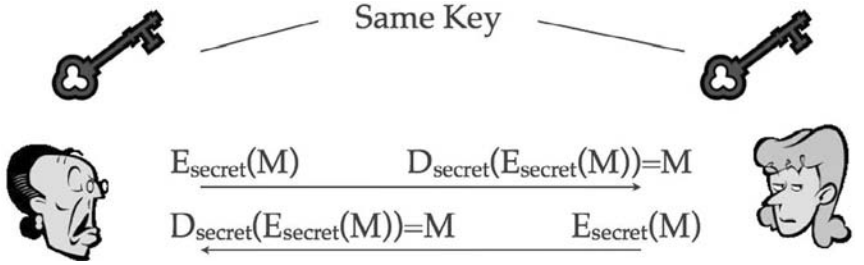


Figure 9.1 Alice and Bob use conventional cryptography.

string of bits (typically 128 or 256 bits long) that determines the mapping from plaintext to ciphertext for encryption and from ciphertext to plaintext for decryption, an attacker can always try to break the encryption by trying all of the possible keys. This is called a **brute force attack**. In order to defend against this attack, we try to use a key space large enough such that the attacker can never, even with large networks of fast computers, try all of the possible keys in any realistic amount of time. Today, a typical key size for conventional cryptography is 128 bits, which yields 2^{128} or 3.4×10^{38} keys. On average, we need to try half of all the possible keys before we will find the correct one using a brute force attack. If we deployed a million computers, each trying a million keys per second, it would take on average 5.4×10^{18} years to find the key needed to decrypt the message. This is about a billion times the estimated age of the universe!

All of the security in conventional cryptography depends on only two things: a sound algorithm (assumed to be known to all) and a long, randomly generated key kept secret between communicating parties. Of course, it is possible that the brute force attack could find the key in a shorter than average time (or take longer than the average time). Since we can never prevent these attacks against the key space, we must use an algorithm that has no weakness that would allow an attacker to find the key or the plaintext more efficiently than brute force.

Let us now consider how we can securely transport the shared secret key from Alice to Bob. We cannot encrypt the key—what key would we use to encrypt the key and how would we get that encryption key to the other party? In practice, we could send the key via some alternative communication channel that is hopefully free from eavesdropping, such as email or the telephone or a person with a handcuffed briefcase, but these are cumbersome alternatives. This form of cryptography (and its cumbersome key transfer methods) is known as conventional cryptography because it was the only type of cryptography known before the mid-1970s (and the invention of **public key cryptography**).

There are two families of conventional cryptographic algorithms: **block ciphers** and **stream ciphers**. Block ciphers encrypt a block of bits, typically 128 bits long, at a time and do not maintain state. This means that each resulting ciphertext block does not depend on the encryption of previous blocks (although we will soon see how to add dependency with a concept called **modes**). Stream ciphers encrypt a bit or a byte at a time while maintaining state from previous encryptions. This means that the same plaintext byte or bit will result in a different ciphertext byte or bit depending on the previous encryption result. These state-dependent mappings help thwart cryptanalysis. Over the years, many conventional cryptographic algorithms have emerged.

With block ciphers (and also, as we will see, with public key cryptography), we need to consider the problem of encrypting data when it does not match the block size

of the cipher. The data we have can be smaller or larger than the cipher block size. We use **padding** to handle the smaller case and **modes** to handle the larger case. If the data we have is smaller than the cipher block size, we must pad the data, which means that we add bits to the data until it matches the block size. Unfortunately, just adding 0s or 1s is not cryptographically secure, so special padding algorithms, usually matched to specific cipher algorithms, perform this function securely. If the data we have is larger than the cipher block size, we must split it up into chunks that match the block size. The options for doing this are called modes. The obvious way to do this is to just take each block size worth of plaintext and encrypt it independently of any previous encryptions. This is known as **electronic code book (ECB)** mode. Unfortunately, this approach creates several problems. Since each block of plaintext is encrypted to the same value ciphertext each time, an attacker can build a dictionary of ciphertext values knowing that they are repeats of the same plaintext values. An attacker can also carry out a replay attack by injecting or rearranging ciphertext blocks, causing unexpected results for the receiver.

A better solution is to chain together successive blocks of encryption so that the encryption of one block is dependent on the previous encryptions. This yields ciphertext for a block that is different from the ciphertext generated for the exact same plaintext block elsewhere in the message. This causes any rearrangement of the encrypted ciphertext blocks to result in an incorrect (and incomprehensible) decrypted version of the plaintext because of the broken dependency chain. Many different modes can accomplish this, with different engineering considerations, but the most common one is **cipher block chaining (CBC)**. When using this mode, we have to start the chain with a value called an **initialization vector (IV)**. The IV does not have to be encrypted, but it should be integrity protected and never be reused.

We will now examine three of the most common conventional algorithms: two block ciphers (DES/Triple DES, AES) and one stream cipher (RC4).

9.1.1.1 DES/Triple DES Historically, the most famous algorithm for conventional cryptography was the **Data Encryption Standard** (**DES**). IBM submitted an internally developed cipher for a National Institute of Standards and Technology (NIST) standardization effort and it became DES, and reigned from 1976 to 2002. DES uses a 64-bit block size and a 56-bit key length. Unfortunately, it is no longer recommended due to the general vulnerability to brute force attacks (but it has no severe breaks). Several factors contributed to this vulnerability, including the 56-bit key length, the increasing pace of CPU speed improvement, and the ease of harnessing large networks to provide parallel computing capacity. A cipher based on DES as a building block has extended its life. By increasing the key length (to either 112 bits or 168 bits depending on the specific implementation), and encrypting data with DES three times in succession, we get an effective strength of about 2^{56} that of DES alone. This approach is known as **Triple DES, 3DES**, or **DESede**. Note that there is no "Double DES" algorithm because an attack called the **meet-in-the-middle attack** is known. This attack weakens the strength of a Double DES approach—and actually degrades it to the same strength as the original DES. This attack also explains why the strength of Triple DES is only 2^{56} instead of 2^{112} times as strong as DES alone. While Triple DES is an improvement, it should be viewed as a stopgap solution because of the performance inefficiencies of using DES three times in succession.

9.1.1.2 AES In 2002, NIST selected an algorithm called **Rijndael** [Daemen02] as the successor to DES and renamed it. The new name they gave it is the **Advanced Encryption Standard (AES)**. AES has a block size of 128 bits and a choice of three key lengths (128, 192, or 256 bits). The criteria for its selection as the replacement for DES included many factors, such as resistance to all known attacks, good performance when implemented in both hardware and software, efficient resource usage (small memory footprint and small CPU requirements for smart card implementations), and fast key changing times. Currently, best practice is to use the AES algorithm in new applications, typically with a 128-bit key length.

9.1.1.3 RC4 Ron Rivest of RSA, Inc. created **RC4** but never officially released it. Ironically, it was leaked to the public, and once it was leaked, it became commonplace in many applications, ranging from wireless network encryption to browser security via Transport Layer Security (TLS)/Secure Sockets Layer (SSL). RC4 is a stream cipher that encrypts and decrypts one byte at a time. It uses a shared secret key as the seed for a pseudorandom number generator. At the sender side, each successive byte of the random stream is Exclusive-ORed (XORed) with each successive byte of plaintext to produce each successive byte of ciphertext. At the receiver side, each successive byte of the regenerated random number stream (which is the same as the sender's stream because of the common key/seed) is XORed with each successive byte of ciphertext to recreate the original plaintext byte.

There are several bitwise operations (meaning that the operation is conceptually done on a single bit but logically can be extended to a byte, or any number of contiguous bits) defined for computers, but here we are using the "Exclusive OR" (XOR), which takes two binary operands and returns a single binary result. There are only four possible combinations to consider: the first operand (A) is a bit that is either a zero or a one, and the second operand (B) is also a bit that is also either a zero or a one. The XOR operation returns a zero when the bits in the two operands are the same and returns a one when the bits in the two operands are different. For example, extending this to a sequence of bits, if the first operand was the binary number 1100, and the second operand was the binary number 1010, the result of 1100 XOR 1010 would be 0110. Notice that the operation is commutative: in other words, 1010 XOR 1100 also results in 0110. Also, XORing A to B and then XORing the result to B again yields A once again, which is an important property for its use in cryptography.

XOR Operation Summary

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

This approach takes advantage of two properties of XOR. First, if we XOR the same bits twice (in this case, the pseudorandom number stream), the result will be the same as the original data. Second, if we XOR with random bits, then the result (in this case, the ciphertext) will also look like random bits. RC4 can be viewed as a pragmatic implementation of a theoretical cipher known as a **one time pad** (**OTP**). When using an OTP, the sender XORs the plaintext with the key (a string of random bits, which is exactly the same length as the plaintext), thereby producing the ciphertext, which is then sent to the receiver. The receiver receives the ciphertext and then XORs it with the same shared key (that string of random bits used by the sender), thereby producing the plaintext. OTPs are impractical because the key lengths need to be as long as the plaintext, the shared key must be perfectly random, and it can never be reused. RC4 makes this idea more practical by generating the key using a pseudorandom number stream. This stream can be exactly as long as needed and uses a shared secret seed value, which can then be viewed as the conventional shared secret key.

9.1.2 Message Digests and Message Authentication Codes

Ensuring data **integrity** is a fundamental requirement for any database. We must guard against both accidental database modification and malicious attempts to modify the data. Simple error detection and correction algorithms, such as cyclic redundancy codes (CRCs), which were designed to detect errors due to noise and other communication faults, cannot be used for security. If we did use one of these algorithms, an attacker could easily modify both the data and the CRC designed to protect the data in such a way that the receiver would be unaware of the data modification. There are two classes of cryptographic building blocks designed to ensure data integrity, message digests (MDs) and message authentication codes (MACs). MD algorithms take a message of unlimited size as input and produce a fixed sized (often several hundred bits long) output. The output is often referred to as a **hash** of the input. The algorithm used for MD is designed to quickly process the large number of input bits into the fixed sized output. It is essential that this function is one way, meaning that it is computationally infeasible to reverse engineer the input bit stream from a given output hash value. This means that an attacker cannot modify the data to match the MD value in an attempt to fool the receiver into thinking that the data still has integrity. MACs operate similarly to MDs, except that they use a shared secret key to control the mapping from input message to output hash. Because they use a secret key, MACs can also authenticate data origin.

9.1.2.1 MD5 and SHA MD5 was the most popular MD algorithm. It generates a 128-bit output hash. But, because of weaknesses identified with the algorithm, it is no longer recommended. Despite these weaknesses, it does still exist in many legacy implementations. Its replacement, an algorithm called **SHA-1**, has a 160-bit output. However, weaknesses were found in SHA-1 as well, so it also is no longer recommended (and it also still exists in many legacy applications). A stronger form of SHA, called **SHA-256**, is currently recommended while a NIST-sponsored effort is underway to develop a replacement standard MD algorithm. A popular framework for turning an MD algorithm into a MAC algorithm is called **HMAC (hash message authentication code)**. Common versions include HMAC-MD5, HMAC-SHA1, and HMAC-SHA256, which are based on the named MD algorithms.

9.1.3 Public Key Cryptography

The biggest drawback of conventional cryptography is the requirement for a shared secret key (and the awkward or insecure options for distributing that key). In the mid-1970s, a number of solutions to this problem appeared. All of these solutions extended the notion of a single key to a key pair, consisting of a **private key** (similar to the secret key of conventional cryptography) and a mathematically related partner called a **public key**. The public key can safely be revealed to the world. The public key can be sent to any party that wants to send a secret message to the key's owner without any communication channel protection or prior prearranged secret. Consider the example shown in Figure 9.2. Whenever Bob wants to send a secret message to Alice using public key cryptography, he first needs to have a copy of Alice's public key. Because this is a public key, Alice can make her public key available to Bob using any mechanism she wants to use (including any nonsecure mechanism). Once Bob has Alice's public key, he can encrypt the plaintext of the message he wants to send using Alice's public key, and then send the resulting ciphertext to Alice. The ciphertext can also be sent using any mechanism, since it is encrypted. Once Alice receives the ciphertext message from Bob, she can decrypt it using her private key. Because Alice is the only one with her private key, she is also the only one who can decrypt messages encrypted using her public key. If she wants to send a message to Bob, she would do the same thing Bob did (obtain a copy of Bob's public key, use it to encrypt the message, and send it to him). With this technology, any two parties can ensure the confidentiality of their communication without previously exchanging any secret information.

Unfortunately, public key cryptography algorithms are much less efficient than conventional cryptography algorithms, typically by several orders of magnitude. This means that it is usually not practical to encrypt or decrypt long messages with public key cryptography. However, we can use it to encrypt a secret key. Then, we can simply send the encrypted secret key using a nonsecure communication channel. Now, we can use conventional cryptography without the awkward secret key distribution issue. By using this hybrid approach, public key cryptography is used for its strength (privately distributing a secret key without prior secret sharing) and conventional cryptography is used for its strength (speed).

We will now look at two algorithms for public key cryptography: the **RSA** algorithm and the **Diffie–Hellman** algorithm. There is also a new family of algorithms called

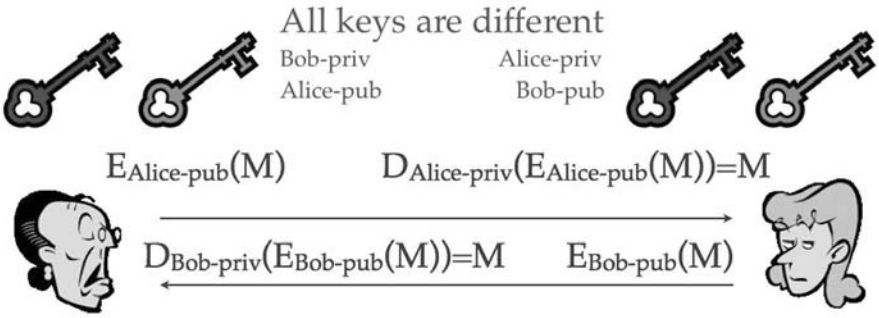


Figure 9.2 Alice and Bob use public key cryptography.

elliptic curve cryptography (ECC), which might emerge in the future, but we will not cover them here. Note, as previously discussed, we must pad data that is smaller than a block size and choose a mode for data that is larger than a block size.

9.1.3.1 RSA and Diffie–Hellman

Although there are a few different algorithms that can be used for public key encryption, we will only mention two of them. The RSA algorithm [Rivest78] is one of the oldest yet still the most popular public key cryptographic algorithm. It is based on the computational difficulty of factoring large numbers into component prime factors. Typically, a user generates a key pair of size 1024 bits (although the best practice is moving to 2048 bits). The RSA block size is typically the same as the key length.

The Diffie–Hellman algorithm [Diffie76] does not directly perform encryption. It is an algorithm run between two parties. Each party performs a calculation based on secret information. This secret information is not preshared with the other party. Instead, the two parties exchange some public results of those calculations with each other and then perform another calculation. The algorithm ensures that both parties will have calculated the same result, which can then be treated as a secret key and used with a subsequent conventional cryptographic algorithm such as AES. Diffie–Hellman is an example of a class of algorithms known as key agreement algorithms.

9.1.4 Digital Signatures

Digital signatures (DSs) are an authentication technique based on public key cryptography. A DS can provide authentication, just like a MAC can, but a DS can also provide a more advanced function, namely, nonrepudiation. There is a subtle distinction between authentication and nonrepudiation. Suppose Alice receives a message from Bob, and that message has an associated and appended MAC. Also, suppose that Alice and Bob had previously obtained the secret MAC key. Alice can then take her key and run the message through the MAC algorithm to verify that she generates the same MAC code as the one appended to Bob's message. Because Alice and Bob are the only ones with the shared secret key, Alice knows that only Bob could have originated this message and associated MAC. Therefore, Alice has authenticated Bob as the source of the message (and authenticated the integrity of the message). But, what would happen if Bob denied that he ever sent Alice this message? If Alice took Bob to court, could Alice prove that Bob originated the message? Although Alice knows that Bob is the only one who could have sent this message (because Bob is the only one who had the other copy of the secret key), Bob could claim that since Alice also has a copy of the secret key, she could have composed the message and sent it to herself! Alice would not be able to prove him wrong, so although MACs provide authentication, they do not provide nonrepudiation.

With a DS, we can use the public and private key pair and certain public key cryptographic algorithms (RSA is the most popular choice) to perform a completely different function than the encryption function we've already discussed. When Bob wants to send a signed message to Alice, he uses his private key to "encrypt" or sign the message. While this action uses an encryption algorithm, it isn't true encryption because anyone with the corresponding public key can decrypt the message, and anyone can potentially get access to the public key. While the message is not encrypted, it is authenticated, because the message receiver can use Bob's public key to "decrypt" or

The most common solution for this problem is to employ a **digital certificate**. A digital certificate for a user or machine consists of a public key, some identifying information (name, location, etc.), and some dates indicating the certificate validity period. All of this information is unencrypted, but it is signed with the private key of a trusted third party known as a **certificate authority (CA)**, such as Verisign, Inc. For a fee, the CA will verify the identity of a party requesting a certificate and then sign the certificate with their private key. The certificates are in a format defined in a standard called X.509. The CA public key is distributed in browsers and operating systems. Using this technology, users do not send their public keys directly to another party. Instead, they send their digital certificates. Certificate receivers can then verify the CA signature using their own copy of the CA public key (in their browser or operating system). This authenticates the owner of the public key contained inside the certificate, which can then be used for subsequent conventional key distribution, encryption, or digital signature verification. If a security problem with the key or certificate is found, then the certificate serial number is placed on the CA's **certificate revocation list (CRL)**, which should always be checked prior to using any certificate. Most browsers have a setting that automates this checking, but for performance reasons, the default is usually set to not check the CRL.

Note that digital certificates do not need to be signed by a CA, but can be self-signed or signed by a company for its own use. This works well for internal applications (and there is no external CA cost). External web users, however, must be presented with a CA-signed certificate or they will see a message box in their web browser indicating that the certificate cannot be trusted.

9.2 SECURING COMMUNICATIONS

DDBEs require secure communications, both between users and the DDBE and among the Sub-DBEs themselves. Secure communication typically demands privacy, authentication, and integrity. This means that we must protect against both passive eavesdroppers who can monitor messages passed over the network and active attackers who can not only monitor messages but can also inject new or modified messages into the network. In general, whenever we need to implement some security capability, we should use an existing, well-hardened protocol. This protocol should, in turn, use one or more of the existing, well-hardened cryptographic building blocks. We should use building blocks that were written by an experienced vendor or a trusted open-source project. Security algorithms, protocols, and implementations are extremely hard to design, implement, and deploy, and history is littered with well-intentioned, failed attempts. We will look at two well-known, well-hardened technologies for achieving end-to-end secure communications: **SSL/TLS** and **Virtual Private Networks (VPNs)**.

9.2.1 SSL/TLS

The most common protocol for securing communications is the Secure Sockets Layer (SSL) protocol, which is now known as the Transport Layer Security (TLS) protocol. This protocol, originally developed by Netscape, makes use of all of the cryptographic building blocks we have discussed. Here is a conceptual sketch of how this protocol works. Suppose Bob wants to communicate with his bank's web server via a web

browser. The TLS protocol provides two services in this scenario: it provides end-to-end communication link privacy via encryption and it provides an authentication service that allows Bob to authenticate the identity of his bank's server via digital signatures and digital certificates. Bob begins by opening a Transmission Control Protocol (TCP) connection from his browser to his bank's server, as shown in Figure 9.3. Usually we want to use a **Uniform Resource Locator (URL)** that begins with "https://," which will attempt to open port 443 on the bank's web server. Using this port indicates a desire to use TLS communications instead of the normal, insecure communications, which uses a URL beginning with "http://" and usually uses port 80. Note, however, that some web servers are configured to treat "http://" requests as "https://" requests.

After the TCP connection is established, Bob's browser and the bank's server negotiate a cipher suite, indicating the specific algorithms that they will use for subsequent communication. For example, RC4 might be used for the conventional cryptography, RSA for the public key cryptography, and SHA-1 for MD. After this negotiation, the bank's server sends its digital certificate to Bob's browser. This digital certificate is signed by a CA and contains the bank's server name, public key, validity dates, and other information. Bob's browser then verifies this certificate using the CA's public key, which was preinstalled in Bob's operating system or browser. Bob's browser can then verify that the machine he is communicating with matches the one named in the certificate, which authenticates the bank's server to Bob. Now that Bob's browser has an authenticated public key for the bank's server, his browser can generate a conventional cryptographic key and encrypt it using this public key. Once the conventional key is encrypted, Bob's browser sends it to the bank's server. Finally, the bank's server decrypts the conventional key that Bob's browser sent, and they can begin to communicate privately (using RC4 in this example).

There is an optional TLS protocol phase that also allows the server to authenticate the client, but it requires a client digital certificate so it is rarely found in consumer scenarios like the one we just described. This optional phase is useful for securing dedicated TLS links between two machines. When the TLS protocol is used without a web browser for these purposes, it is often called TLS secure socket communication. Note that TLS is a form of end-to-end security, meaning the entire communication path from client to server is protected. This means that every message sent using this protocol is protected, even when they pass through insecure links along the way, such as an unencrypted wireless network.

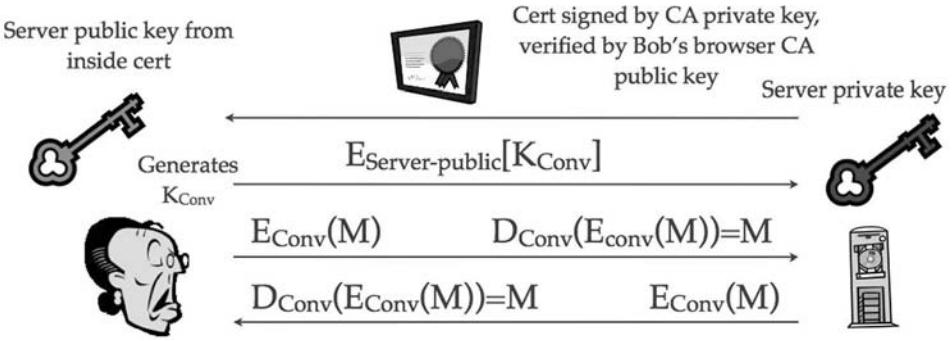


Figure 9.3 Bob uses TLS/SSL to connect to his bank's server.

9.2.2 VPN, IPSec, and SSH

Briefly, here are three other examples of technologies (like TLS) that can be used to provide end-to-end secure communications. One approach to securing communication links is to use a technology called **Virtual Private Networks (VPNs)**. This technology was originally designed to allow the insecure Internet to be used securely with an additional layer providing security services. This is useful for both remote worker-to-business and site-to-site security, such as between DDBE components. Another approach is to use a protocol called **IPSec**, which provides both encryption and digital signature services to the IP layer of the Internet. This allows all of the higher layers (TCP, UDP, and applications) to benefit from this security service. The IPSec protocol is optional for IP Version 4, but it is mandatory for the next generation, IP Version 6. Often, VPN services are built on top of IPSec. **Secure Shell** (SSH) is a similar authentication/encryption protocol that can also be used to build secure tunnels between two endpoints.

9.3 SECURING DATA

While secure communication between users and DDBEs, and between DDBE components themselves, is necessary for overall system security, it is not always sufficient. Sometimes, we must also protect the data residing in the databases from unauthorized viewing and modification. Even when these security measures are in place, we can be vulnerable to various other attacks. In this section, we will begin by examining some techniques for authorizing data access. Then, we will consider techniques for encrypting data. Next, we will look at two classes of database-specific security attacks. The first class of attacks (unvalidated input and SQL injection) depends on deployment and implementation details, while the second (data inference) has more subtle causes. We will close this section with an overview of data auditing.

9.3.1 Authentication and Authorization

Authentication for both DDBE users and components can be implemented using digital certificates, as described above in the TLS protocol using client certificates, or with user ID and password. Security tokens (in the form of a key fob or a credit card-sized device) are increasingly popular extensions to the user ID/password approach. They display a sequence of digits, which randomly change every 30 seconds or so. To authenticate, the user enters his/her user ID, the displayed digit sequence, and his/her PIN, creating a one-time password. This is known as two-factor authentication: something you have (the token) and something you know (the PIN). Biometrics offers a third factor to use for authentication: something you are (such as a fingerprint).

After users are authenticated to the DDBE, they can only access resources if they are authorized to do so. If we model our DDBE security service on the traditional Relational Structured Query Language (SQL) approach, we can offer authorization control via the GRANT statement. If we execute the GRANT command on the left in Figure 9.4, the Alice DDBE Account will be granted the right to insert and delete rows in the Employee table. If we execute the GRANT command on the right in Example 9.1, the Alice User Account will be granted the right to read rows from the Employee table

```
GRANT INSERT, DELETE
ON    Employee
To    Alice;
```

```
GRANT SELECT
ON    Employee
TO    Alice
WITH  GRANT OPTION;
```

Figure 9.4 Granting the Alice DDBE User privileges on the Employee table.

and her account will also be granted the right to extend this authorization (SELECT) to other DDBE Accounts.

A technology called **Lightweight Directory Access Protocol (LDAP)** can also support authentication and authorization. Within this context, a directory is a collection of generic information (such as white pages listing people, their phone numbers, addresses, email addresses, etc.) Directories can also contain authentication information such as user ID and password information. Similarly, directories can contain authorization information that specifies what software an authenticated user can or cannot access. By using either a single LDAP server or a collection of cooperating LDAP servers, administrators can use a single facility to control distributed authentication and authorization details. This strategy is called **single sign on (SSO)**. Microsoft Windows Active Directory provides a similar capability. We must secure all the communication links to LDAP servers too, which is often done using TLS.

Firewalls, often specialized for use with databases, can control access at the network level. Administrators can specify parameters such as a list of IP addresses, ports, protocols, dates, and times for which accesses are allowed or denied. This enables the firewall to filter unauthorized network traffic before it arrives at the DDBE components. **Intrusion detection systems (IDSs)** can monitor network traffic for patterns of malicious traffic (such as viruses, worms, or live attackers) that can potentially compromise a database. The IDS can then alert an administrator to take action. Note that IDSs can be human resource intensive because they typically generate many false positives.

9.3.2 Data Encryption

If the physical hardware systems used by our DDBE components are ever physically compromised, the disks could be removed and scanned for sensitive data. Even if they are never physically compromised, these systems could be attacked across a network from remote machines, from local machines on the network, or even from processes running directly on the host server itself. This means that sensitive data should be encrypted inside the database storage.

Encrypted file systems (EFSs) are becoming increasingly popular for protecting hard disk drives (especially on laptops) containing sensitive data. All the data on the disk is encrypted with a conventional key and accessed via a user password. However, this technology can often create performance problems when used with database systems because database system designers have taken great care to optimize data access assuming specific physical disk layouts that might be disrupted when data is encrypted.

A better approach for encrypting database data is to encrypt the data being stored rather than the underlying file system. There are two basic approaches: internal to

the DBE, and external from the DBE. In the first approach, we use components or applications to encrypt the data prior to sending it to the DBE for storage and then use similar components or applications to decrypt the data after it is retrieved from the DBE. Since the DB is storing the encrypted version of the data, the data types used inside our DBs must be modeled to hold the encrypted ciphertext, not the plaintext. This often requires extensive use of binary data types in the database, such as variable-length binary (varbinary) types or binary large objects (BLOBs). In the second approach, because the DBE directly supports encryption, the DDBE users can simply read and write their data normally while the DBE handles the encryption "under the covers." This is generally the most efficient approach to data encryption for centralized DBMS.

With any type of encryption, we must be careful with key handling, because a compromised key means compromised data. Often, the key is protected by a password. This password must be stronger than the key it is protecting. In this context, stronger has a very specific, technical meaning, called **entropy**, that refers to more than just the literal password length. A long password consisting of words easily found in a dictionary and using only a subset of possible characters (such as all lowercase letters) has lower entropy than a password that is the same length but uses random, uniformly distributed characters.

Users must never store their passwords directly on a machine unless the passwords are also encrypted using a sufficiently strong encryption mechanism. Software providing password vaulting, a technique where a master password is used to decrypt a list of other passwords, is a reasonably safe way to store passwords. The master password must be at least as strong as the contained passwords. The Mac OS X keychain is an example of a password vault.

We must also be careful when the password is supplied by a machine or process instead of a person. Hard-coding passwords in command scripts can make them vulnerable to both snooping and to system utilities that list the command lines of running processes. Hard-coding passwords in source code can make the passwords vulnerable to string extraction utilities. The best options use operating system provided key-protection constructs such as keychains and ACL protected registries. In the future, secure hardware module key storage will probably become increasingly popular.

9.3.3 Unvalidated Input and SQL Injection

Unvalidated user input is one of the most common software security flaws. It is responsible for exploits ranging from buffer-overruns, to command injection, to cross-site scripting. It is also responsible for a class of exploits specific to databases known as **SQL injection**. This attack takes advantage of a lack of user input validation, usually in web-based input fields for forms and in URLs containing post data. The attack uses cleverly crafted input values to sneak in SQL commands that can expose confidential information, allow unauthorized data modification, or even destroy or corrupt the data in the database.

Suppose we have a web-based application that uses web-forms for user input, which it then uses to build SQL commands dynamically. It creates these SQL commands at runtime by concatenating SQL command fragments and the user-supplied input values. Although the actual DDBE authorization and authentication would **never** be based on this user input, this is an easy example to explore. For each of these scenarios, suppose our web-based application is an online storefront of some kind,

and the web-form in the scenario is the customer login and password entry form. Assume we have a relational table named "CustomerLogin" in the DDB that contains a unique "CustomerId" value for each unique pair of "CustomerName" and "CustomerPassword" values. Our web-form must have two user-supplied data fields; let's call them "inputName" and "inputPass." Now let's consider what happens when this form is attacked using a SQL injection technique.

9.3.3.1 Bypassing the Password for Customer Login Suppose our application dynamically builds a SQL Select statement using the hard-coded strings shown in Figure 9.5, and the user input values. Notice that String2 and String3 have a leading space. Both String3 and String4 have a single-quote (apostrophe) at the end, and both String4 and String5 have a single-quote (apostrophe) at the beginning. String5 ends with a semicolon, which is the "end of command" character for SQL. When the customer logging in provides values via the input fields in this form, our application creates a SQL query by concatenating String1, String2, and String3 together, followed by the value of inputName field, then String4, followed by the value of inputPass, and then, finally, String5. This SQL query will return the CustomerId value from the relational table where the CustomerName value and CustomerPassword value each match the values specified in the web-form fields.

For an example of what we intended to have happen, if a customer named Alice used this form to enter her name and password (let's suppose she enters the value "Alice" for inputName and "Rabbit" for inputPass), our application would create the SQL query in Figure 9.6. If the specified values (CustomerName equal to "Alice" and CustomerPassword equal to "Rabbit") do **not** exist for any row in the table, then the SQL query will return an empty set (zero rows returned, a "SQL-Not-Found" error raised, etc.). If this happens, since there is no "CustomerId" value returned for this query, the login attempt will fail. On the other hand, if the SQL query returns a nonempty result set, then it will contain Alice's CustomerId value, and the login attempt will succeed.

For an example of an unintended consequence, if we have "unvalidated" input for this web-form (if our code does not adequately check for malicious or invalid customer input), it is possible that some attacker could cleverly construct values for these input

```
String1 = SELECT CustomerId
String2 =  FROM CustomerLogin
String3 =  WHERE CustomerName = '
String4 = ' AND CustomerPassword ='
String5 = ';
```

Figure 9.5 Hard-coded strings used to build SQL query.

```
SELECT CustomerId
FROM   CustomerLogin
WHERE  CustomerName = 'Alice'
AND    CustomerPassword = 'Rabbit';
```

Figure 9.6 SQL query built for customer "Alice" with password "Rabbit."

fields to circumvent both our authentication and authorization mechanisms. Suppose our attacker provides the input values shown in Figure 9.7. The value for each field is the same: a single-quote, followed by a space, the word "OR," and another space, and then two single-quotes followed by an equal-sign and another single-quote. What will happen next?

Figure 9.8 shows the SQL query that our application would create and execute in this scenario. Notice that the single-quotes changed the logic of the WHERE clause. Now, we will return the CustomerId value wherever the CustomerName value is empty OR wherever an empty string equals an empty string. This would effectively return a random CustomerId value and allow the attacker to be logged in as this random customer. By specifying a known customer name for the inputName field, along with this same malicious password value, an attacker can impersonate any customer.

9.3.3.2 Bypassing the Password for Change Password This same technique can be used to circumvent other unvalidated web-forms, even when we attempt to enforce security. For example, suppose our attacker has logged in using CustomerName "Alice" and obtained her CustomerId as shown in Figure 9.8. Suppose Alice's CustomerId value is "1234." Now, suppose the attacker wants to change Alice's password—effectively preventing Alice from being able to log in to her own account. Suppose we had another (unvalidated) web-form for changing passwords. Even if this form attempted to provide additional security by requiring the customer to enter the old password value along with the new, the attacker could again use the technique (and values) shown in Figure 9.8. Here, the system would provide the CustomerId value, and our attacker would provide the "old CustomerPassword value" (same as shown in Figure 9.7), and a "new CustomerPassword value" (suppose the new password is "newValue"). Figure 9.9 shows how the exploited SQL Update command might look in this scenario.

9.3.3.3 Vandalizing the DDBE In the previous examples, our attacker was attempting to impersonate one of our customers. Suppose we encounter an attacker who wants

```
inputName =' OR ''='
inputPass =' OR ''='
```

Figure 9.7 Malicious input values entered by an attacker.

```
SELECT CustomerId
FROM   CustomerLogin
WHERE  CustomerName = '' OR ''=''
AND    CustomerPassword = '' OR ''='';
```

Figure 9.8 SQL query built from malicious input (shown in Figure 9.7).

```
Update CustomerLogin
SET    CustomerPassword = 'newValue'
WHERE  CustomerId =1234
AND    CustomerPassword = '' OR ''='';
```

Figure 9.9 SQL update built from malicious input.

to vandalize our website and DDBE. There are several scenarios where the same SQL injection technique can allow such an attacker to do very damaging things to our system. The issue of unvalidated input again plays a key part in this security breach, but we also have a new class of exposures that increases the potential for harm based on the violation of the least privilege principle, or granting more authority than necessary.

Although there are several different approaches for identifying the right level of authority for a given application or scenario, they are beyond our scope here. Instead, we will focus on a simple example demonstrating what can happen when we are too generous with the authorization policy. When our web application connects to the DDBE, the DDBE authenticates the Account information that the application is using. Whenever the application attempts to execute a query or command, the DDBE verifies that the account has the authority to perform the operations required for request. For example, the account used by our application must have the ability to read the CustomerLogin table if it is going to verify customer names and passwords, and the ability to update the CustomerLogin table if it is going to change customer passwords.

Suppose this account is authorized to do more than simply read and update a single table. In the previous examples, the attacker tricked our application into executing clever variations of the intended commands. In this scenario, the attacker uses the same security vulnerability (unvalidated input) and the same exploit (SQL injection) but, because we have given a tremendous amount of authority to the underlying DDBE account being used by our application, the unintended commands will be more destructive. For example, suppose our attacker uses the same web-form and the same dynamic SQL scenario as we depicted in Figure 9.5, but now, the values entered are those in Figure 9.10.

Notice that the inputName starts with a single-quote, and then is followed by a semicolon. Recall that the semicolon terminates a SQL command, which means that the remaining text is a new command. The remaining text in the name value is a command to drop the CustomerLogin table. If the account is authorized to do this command, it will destroy the CustomerLogin table structure and content! The second semicolon in this field is followed by a space and two dashes (hyphens, minus signs). The space followed by two dashes is recognized as a "begin comment" command in most SQL parsers. Figure 9.11 shows the SQL command generated for this scenario. Everything after the two dashes is ignored. The fact that the SQL query returns no rows is irrelevant, because after the query executes, the entire table will be dropped.

```
inputName ='; DROP TABLE CustomerLogin; --
inputPass =anything
```

Figure 9.10 Even more malicious input values entered by an attacker.

```
SELECT CustomerId
FROM   CustomerLogin
WHERE  CustomerName = ''; DROP TABLE CustomerLogin; --'
AND    CustomerPassword = 'anything';
```

Figure 9.11 Command built from input values shown in Figure 9.10.

This means that customer login information will be lost, and nobody will be able to log into the system.

9.3.3.4 Preventing SQL Injection There are several techniques for thwarting this attack. First, we can perform simple input validation, such as ensuring CustomerName values can only contain alphanumeric data (no spaces, punctuation, numbers, etc.). Similarly, verifying that the user-supplied value falls between the minimum and maximum lengths defined for the field can limit the amount of malicious content. Many RDBMS platforms provide a facility known as "prepared SQL," or "parameterized SQL;" facilities like these can use precompiled type checking as a technique to minimize exposure to attacks such as SQL injections. Unfortunately, there is no such thing as a commercial off-the-shelf (COTS) DDBE product. This means that we must manually create the facilities used to parse and execute commands.

SQL injection is just one example of a broad class of software flaws that can compromise security in applications. In order to guard against these attacks, there are many preventative actions that we need to perform, such as ensuring that software memory buffers on stacks and heaps are not overrun in our components, subsystems, and applications, protecting against injection for operating system commands and DBE commands, and properly handling exceptions and return codes. For a good exploration of these application security issues see [Howard05].

9.3.4 Data Inference

Databases can also suffer from a specific security and privacy vulnerability, known as a data inference attack. This attack is extremely difficult to address, and impossible to completely solve as a general case. When we allow certain, individual queries that are by themselves innocuous, we cannot prevent the aggregation of these results, which then exposes confidential information. It is a specific example of a more general problem: when attackers can create data mining results from disparate sources, each of which is authorized, in order to gain confidential results through the aggregate. Suppose we have a relational table named EmpCompensation, containing the columns and data needed to capture employee salary (see Figure 9.12).

Suppose we have three employees named "John," "Mary," and "Karen" (as shown in the figure data), and each employee has an authenticated DDBE account. Employee accounts have the privileges necessary to look up their own salary information, but not the information for other employees. We also allow employees to retrieve statistical information about salaries in the organization (so that they can see where they fall on the salary grid).

EmpName	EmpTitle	EmpSalary
John	Worker	$10,000
Mary	Worker	$20,000
Karen	Manager	$30,000

Figure 9.12 The EmpCompensation table.

- *Phase 2.* The originating site will first send messages to read everything it needs. Then it will send messages to write everything it needs to write. This phase will cost 14C—10C to read data items at Sites 1, 2, 6, 7, and 8; and 4C to write data items at Sites 3, 5, 6, and 7.
- *Phase 3.* The originating site will validate the SI for transaction T. This phase will cost 2C since all SI data items are stored at Site 8.
- *Phase 4.* The originating site will unlock the sites it needs to unlock. This phase will cost exactly the same as Phase 1 (7C) since all data items locked in Phase 1 need to be unlocked.

3.6 SUMMARY " Chapter 3"`

The issues with database control are some of the most challenging issues that a DBA must address. Controlling the database requires implementing tight security measures and correct security policies that define the access rights for users and guarantee the security of the database. This is vital to ensuring the database's consistency and its accessibility only by authorized users. Semantic integrity rules are defined when the database is designed and enforced by the DBMS when the database is used. Semantic integrity rules can be applied before a transaction runs, during a transaction's execution, or after a transaction's execution (but before it has been committed) with each approach having some drawbacks and some benefits.

3.7 GLOSSARY

Access Control The action of controlling who can access what contents of the database.

Access Rights The set of action privileges that are given to a user (or a group of users).

After Trigger A trigger that runs after the triggering event has been run.

Assertions A set of conditions/rules that are defined for the contents of the database.

Authentication Any technique used to validate the identity of a source or destination of messages or data.

Before Trigger A trigger that runs before the triggering event has been run.

Compile Time Validation An approach to semantic integrity control that checks transactions before they run.

Constraints A set of conditions/rules that are defined for the contents of the database.

Data Type A constraint associated with the type of a data item in a database.

Database Administrator (DBA) A person who has ultimate privileges on what can be done to the database.

Integrity Constraints A set of conditions/rules that are defined for the contents of the database.

Postexecution Time Validation An approach to semantic integrity control that checks transactions after they run.

Referential Integrity The fact that for every foreign key value in a table there must be a corresponding primary key value in the database.

Relation Constraints A set of rules that apply to the table as a whole.

Relational Constraints A set of constraints that are inherited by the relational model from the ER (conceptual) model of the database.

Role A grouping of privileges that can be assigned to a database user or another role.

Row Trigger A trigger that runs as many times as the number of rows affected by the triggering event.

Run Time Validation An approach to semantic integrity control that checks transactions during execution.

Semantic Integrity Rules A set of conditions/rules that are defined for the contents of the database.

Statement Trigger A trigger that runs only once regardless of how many rows are affected by the triggering event.

Stored Procedure A procedure that is stored in the database.

Trigger A stored procedure that can only be called by the DBMS when modifying events such as insert, delete, and update are issued by the database users.

User Defined Data Type A data type that a database user can define, which is based on the base data types of a DBMS.

REFERENCES

[Badal79] Badal, D., *The Proceedings of the Fourth Berkeley Conference on Distributed Data Management and Computer Networks*, pp. 125–137, August 1979.

[Eswaran76] Eswaran, P., *Specifications, Implementations and Interactions of a Trigger Subsystem in an Integrated Database System*, IBM Research Report RJ 1820, November 1976.

[Hammer78] Hammer, M., and Sarin, S., "Efficient Monitoring of Database Assertions," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 38–48, Dallas, TX, June 1978.

[Machgeles76] Machgeles, C., "A Procedural Language for Expressing Integrity Constraints in the Coexistence Model," *Modeling in Database Management Systems*, pp. 293–301, edited by Nijssen, G., North Holland, Amsterdam, 1976.

[McLeod79] McLeod, D., *High Level Expression of Semantic Integrity Specifications in a Relational Database System*, MIL Tech Report TR-165, September 1979.

[Stonebreaker74] Stonebreaker, M., *High Level Integrity Assurance in Relational Data Management Systems*, Electronic Research Laboratory Memo ERL-M473, University of California, Berkeley, August 1974.

[Stonebreaker76] Stonebreaker, M., and Neuhold, E., *A Distributed Database Version of INGRES*, Electronic Research Laboratory Memo ERL-M612, University of California, Berkeley, August 1976.

EXERCISES

Provide short (but complete) answers to the following questions.

3.1 Assume the cost of sending a message from any site to any other site is C units of time for SI validation. Figure out the cost of each one of the phases of the

following strategy. Also, assume that there are N transactions and, out of these, M will fail the validation.

Phase 1: Control site read-locks and reads all data that needs to be read.

Phase 2: Control site performs the validation.

Phase 3: Invalid transactions are rejected—locks are released for these transactions.

Phase 4: Control site calculates all the new values that need to be written.

Phase 5: Control site write-locks information at all the sites it needs (no ACK for locking is required).

Phase 6: Control site writes where it needs to write for successful transactions.

Phase 7: Control site unlocks at the sites that need to be unlocked.

3.2 Assume the following two tables in SQL Server 2005 where EMP(Dno) is a Fkey that points to DEPT(Dno).

```
EMP  (Eno Integer  not null, Dno Integer not null,
      primary key(Eno))
DEPT (Dno Integer not null, Mgr varchar(15),
      primary key(Dno))
```

(A) Write a stored procedure that when given a Dno checks the existence of it in the DEPT table. The procedure returns 1 if the Dno found and −1 if Dno is not found.

(B) Write a trigger that uses the procedure in part A to validate inserting a row into the EMP table. The trigger checks the existence of Dno in the DEPT table. If Dno is found, then the row is inserted. Otherwise, the trigger aborts the insert.

3.3 Assume the following two tables in Oracle 10 g:

```
EMP (Eno NUMBER(4) not null, sal NUMBER (10,2), Dno NUMBER
(3) not null)
DEPT (Dno NUMBER(3) not null, Mgr varchar(15), Tot_Emp_sal
NUMBER (12,2) not null)
```

EMP(Dno) is a Fkey that points to DEPT(Dno), which is a Pkey.

(A) Write a stored procedure that when given an amount increases the value of the column Tot_Emp_sal with this amount.

(B) Write a trigger that runs when employees are added to the database. The trigger uses the procedure in part A to update the column Tot_Emp_sal in the DEPT table to reflect the fact that the new employees are added to their corresponding departments.

3.4 Consider the bank database discussed in Example 3.6. This database has the integrity requirements that (1) a loan must belong to one, and only one, customer; (2) an account must belong to one, and only one, customer; and (3) a

customer must own, at least, one loan or one account in the bank—otherwise, the customer cannot exist in the database. All branches in the bank are located in either Minneapolis (MPLS) or St. Paul (STP). All accounts belonging to a branch in MPLS are stored in the database in the MPLS server. Similarly, all accounts belonging to a branch in STP are stored in the database in the STP server. Smith is a customer with $CID = 100$. Smith decides to close the account with A# = 222 by issuing a delete command from an ATM in LA. Explain the steps required to maintain the integrity of this database when this account is closed.

ability to choose any of the DBEs to perform, for example, a join. In a heterogeneous distributed database system, the mediator does not have the same freedom. In a heterogeneous database system, a local DBE may be able to perform a join and the other may not. Lack of join capability by some of the DBEs (or their wrappers) creates a challenge for the mediator that does not exist in a homogeneous distributed database system.

For example, in a homogeneous distributed database system, the global optimizer that needs to join relations "R" and "S" can simply send R from where it resides to the DBE where S is and ask the DBE server there to process the join. In a heterogeneous database system, this may not be easily achieved. Suppose that the wrapper for the DBE that stores S does not have a join capability (such as a BigBook database) and can only select tuples based on a given predicate. In this case, the mediator has to first retrieve all tuples for R from the wrapper that interfaces to the DBE where R is stored. Once all the tuples of the R relation are received, the mediator iterates through all tuples in R and uses the value of the join attribute of R, one-by-one, to find corresponding tuples in S. The lack of a join capability within the S wrapper, in effect, forces a nested-loop join to be carried out by the mediator. This approach is probably a lot more expensive than having the join be performed by the component DBE, as is the case for a homogeneous distributed database system.

Another set of differences that may exist between the DBEs in a heterogeneous database system are called semantic heterogeneity [Sheth90]. For example, Wrapper 1 may interface to a DBE that uses letter grades {A, B, C, D, F} while Wrapper 2 interfaces to a DBE that uses scores between 1 and 10. When integrating the results from these two local DBEs or when transferring the grade from one DBE to the other, the mediator has to convert scores of 1 to 10 to letter grades and/or letter grades to scores in the range of 1 to 10.

Once the capability differences between the local DBEs have been dealt with, the approach to global optimization for both homogeneous and heterogeneous database systems is very similar. Once the capabilities of the underlying DBEs are encapsulated by the wrapper and are presented to the mediator as a set of enumeration rules, dynamic programming or iterative dynamic programming (see Sections 4.4.2.3 and 4.4.2.4 of this chapter) can be used by the mediator to optimize global queries in heterogeneous database systems [Haas97]. In Chapter 12, we provide more details about integration challenges in heterogeneous database systems.

4.6 SUMMARY "Chapter 4"

In this chapter, we outlined the available techniques for query optimization in both centralized and distributed DBEs. We discussed the fact that query optimization is a complex component of query processing. We provided an overview of relational algebra as an internal language representation used by all commercial DBMSs to process user commands. Access path strategies for different relational algebra operations were also discussed. We pointed out that even for queries of moderate complexity the solution space was large. Approaches to reducing the solution space and dealing with the complexity of the query optimization were outlined. Rules and heuristics for query optimization were investigated. We reviewed dynamic programming, iterative dynamic programming, and greedy methods as three alternatives widely used in query optimization.

Ideas and techniques used in centralized query optimization were then expanded to include distribution. We analyzed approaches that were used for query optimization in homogeneous as well as heterogeneous DDBEs. Most notably, we studied semi-join and dynamic programming for distributed database environments.

4.7 GLOSSARY

Access Path The method by which a table is accessed as part of an overall query processing.

Bit Vector Filter A bit-oriented encryption of the join columns to reduce the communication cost between the sites involved in a distributed query.

Bloom Vector A bit-oriented encryption of the join columns proposed by Bloom to reduce the communication cost between the sites involved in a distributed query.

Catalog A set of system-defined tables that maintain information about structures defined within one or more databases. "Catalog" is another term for "data dictionary."

Centralized Database Environment (CDBE) A database environment in which one computer (server) carries out the users' commands.

Clustered Index An index in which the table rows that are addressed by sorted key values are clustered (adjacent) to each other.

Conditional Join A join that requires a condition that two or more columns of the two tables being joined will have to satisfy.

Cost-Based Optimization (CBO) An optimization approach in which the cost method is used to find the query plan with the smallest cost out of the possible query plans for the query.

Data Dictionary A collection of system tables that hold information about the structures of a database. Data dictionary is also known as the meta-data for the databases in a DBMS.

Data Shipping An execution strategy in which data is shipped to another computer to be used in the operation.

Disk I/O The process of reading or writing from/to the hard disk.

Distributed Database Environment (DDBE) A collection of one or more DBs along with any software providing at least the minimum set of required data operations and management facilities capable of supporting distributed data.

Dynamic Programming A phased approach to query optimization which eliminates suboptimal plans until the optimal plan is arrived at.

Equi-join A conditional join that forces equality across two columns of the two tables being joined.

Global Query A query whose target tables are distributed across multiple computers or database environments.

Greedy Algorithm An optimization approach that limits the amount of disk storage and memory used by dynamic programming by only forming 2-relation plans at each step of the process.

Hash-Join A specific join strategy that uses a hash function to partition the rows of the two tables being joined and then only joins rows in corresponding partitions.

Heterogeneous DDBE An environment in which the DBEs are from different vendors and/or support different data models.

Heuristics-Based Optimization An optimization approach that uses heuristics to eliminate alternatives that do not provide or lead to an optimal query plan.

Homogeneous DDBE An environment in which all DBEs are from the same vendor and/or support the same data model.

Hybrid Shipping A mix of data and operation shipping.

Iterative Dynamic Programming An iterative approach to dynamic programming that limits the memory and storage requirements.

Local Query A query whose target tables are on the same computer or database environment.

Materialization The act of forming and storing the temporary results of a relational operator.

Memory Frame A container for storing a page in memory.

Memory Page A unit of storage allocation in memory.

Modulus (MOD) Function An operation that returns the remainder of dividing two numbers.

Nonclustered Index An index in which the table rows are not forced to be adjacent to each other for consecutive key values of the index.

Nonprocedural Language A language in which the user specifies what is needed and the system determines the process of arriving at the answer.

Nonunique Index A nonunique index allows key duplication in the index.

Normal Conjunctive Form A complex predicate in which the component predicates are tied together by the logical operator "AND."

Normal Disjunctive Form A complex predicate in which the component predicates are tied together by the logical operator "OR."

On-the-Fly The process of feeding the results of one relational operator into another without having to store the results on the disk.

Operation Shipping An execution strategy in which the operation is carried out where the data is.

Partitioning The act of dividing the rows in a table based on a given condition applied to a column of the table.

Probing The act of finding a qualified row based on the value of a column typically using an index.

Procedural Languages A language in which the user specifies the process of arriving at the answer.

Quel A higher level language based on relational calculus.

Query Optimization The process of finding an optimal query execution plan for a given query.

Query Plan A plan that specifies a detailed step-by-step execution strategy for a given query.

Query Processing The process of executing a user's query.

Query Trading A specific query optimization approach that uses negotiation to arrive at an optimal query execution strategy.

EXERCISES

Provide short (but complete) answers to the following questions.

4.1 How many alternatives exist for joining three tables together? How many alternatives exist for joining four tables and five tables together? Can you extrapolate from these numbers to arrive at the number of alternatives that can be used to join N tables together?

4.2 A four-site system has tables R, S, T, and M stored as depicted in Figure 4.35. The user is at Site 1. Assume each row of any table that has more than one column can be sent in one message. You can also assume that a one-column table can be sent in one message. Also assume that the cost of sending each message is C units of time. There are no costs associated with sending commands. What is the cost of "(S JN R) JN (T JN M)" if "S JN R" and "T JN M" are done using semi-join and the final join is done normally? Make sure you show all steps and cost of communication for each step.

4.3 Assume the four-site system shown in Figure 4.36—the table distribution and the query tree that prints all information about the employees who work for the engineering department and have an account in a branch in the city of Edina. Also assume the user is at Site 2. Let's assume that 40% of accounts are in branches in Edina; 50% of employees work for the engineering department; and 10% of employees who work for the engineering department have an account in a branch in Edina. If each row being sent from each site to any other site takes C units of time, what is the communication cost of the optimized query tree (results must be displayed to the user) in terms of C? Make sure you show individual step's cost and the total cost. Assume there is no cost associated with sending the commands.

4.4 In the four-site system shown in Figure 4.37, the user is at Site 1. We need to print all information about all sales persons who have accounts in a branch in the city of Edina. We know that 5% of branches are in Edina; 10% of all accounts

Site 1			Site 2		Site 3			Site 4	
S			R		T			M	
A	B	C	A	D	A	E	F	A	G
1	b1	c1	1	d1	1	e1	f1	1	g1
2	b2	c2	2	d2	2	e2	f2	2	g2
3	b3	c3	3	d3	3	e3	f3	3	g3
4	b4	c4	5	d4	4	e4	f4	5	g4
9	b5	c5	6	d5	5	e5	f5		
					6	e6	f6		
					7	e7	f7		

Figure 4.35 Relations for Exercise 4.2.

Emp (Ename, Sal, D#)
Dept (D#, dname, Budget)
Account (A#, bal, bname, cname)
Branch (bname, bcity)

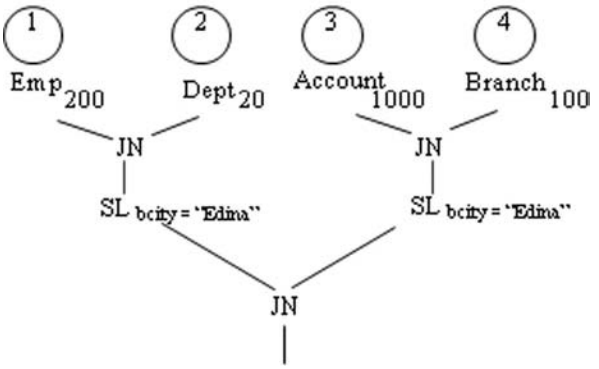


Figure 4.36 Database setup for Exercise 4.3.

Emp (Ename, Sal, D#, Position)
Dept (D#, dname, Budget)
Account (A#, bal, Bname, Ename)
Branch (Bname, bcity)

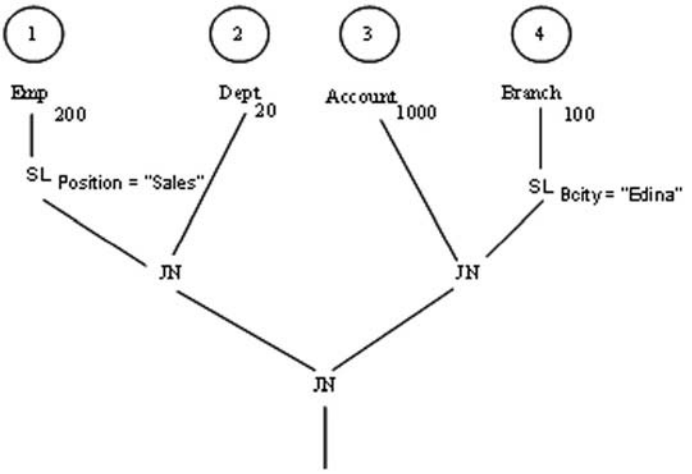


Figure 4.37 Query tree for Exercise 4.4.

Sailors(sid, sname, rating check (1-10), age)
Boats(bid, bname, type)
Reserves(sid, bid, day, rname, desc)

Sailors table stats:
There are 40,000 sailors – 40,000 rows
Each row is 50 bytes long
A page can hold 80 sailor rows
There are 500 pages to sailor table

Reserves table stats:
There are 100,000 reserves – 100,000 rows
Each row is 40 bytes long
A page can hold 100 reserves rows
There are 1000 pages to sailor table

Boats table stats:
There are 100 boats

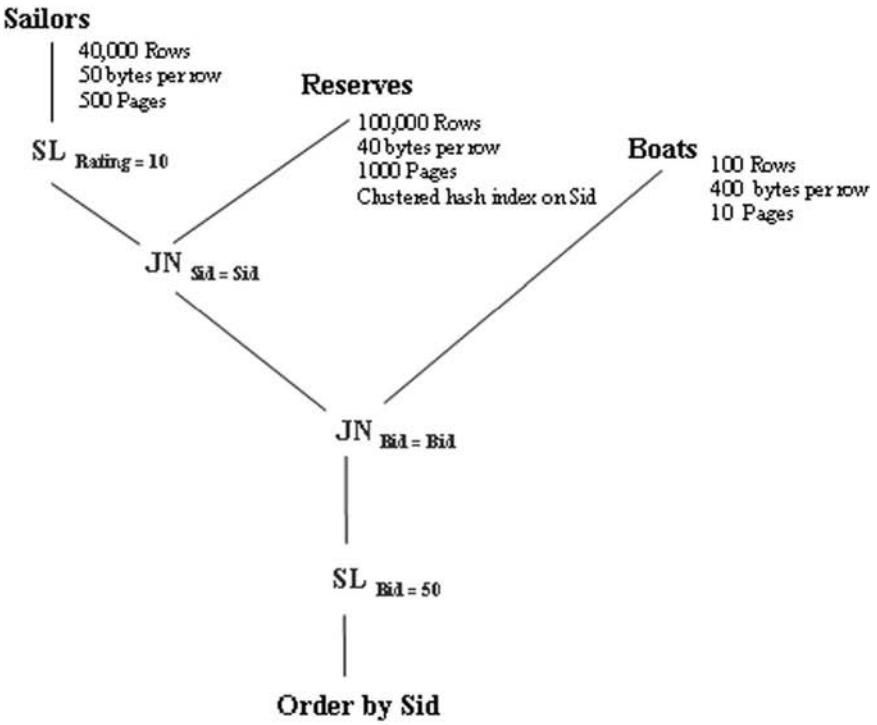


Figure 4.38 Relations for Exercise 4.6.

are in branches in Edina; and 20% of employees are in sales. If processing each row takes "t" units of time, what is the processing cost of this tree? Make sure you show all steps and the cost of each step.

4.5 For the distributed database given in Exercise 4.3, what is the cost of an optimal distributed execution strategy for this query in terms of the number of messages

required? Assume the cost of sending each row of any table is C units of time. Assume there is no cost associated with sending the commands. Make sure you show all steps and the cost of each step.

4.6 For this question, we use a sail club database used in [Ramakrishnan03]. This database has three tables as shown in Figure 4.38. Suppose we have a query that returns a sorted order of the sailors with a rating of 10 who have reserved the boat with ID = 50. The query plan for this question is depicted in Figure 4.38. Assume the page size is 4000 usable bytes after the page overhead. If the first join is done by a nested-loop, row-based join and the second join is done by a sort–merge join, what is the cost of this plan in number of disk I/Os? Assume that we have five buffers for sorting and all distributions are uniform.

5.5 SUMMARY "Chapter 5"

In this chapter, we discussed approaches to concurrency control in centralized and distributed DBEs. The main focus of our discussion was maintaining the atomicity property of transactions. We pointed out that atomicity was maintained by providing the illusion that transactions were running alone in the system. We argued that in order for the concurrency control to provide such an illusion, every transaction's changes needed to be isolated from the other transactions, until such a time as the transaction decided to commit or abort.

Since the focus of our discussion was OLTP systems, we introduced transaction concepts first. The ACID properties for transactions were defined and studied. We categorized two main approaches to concurrency control: pessimistic and optimistic concurrency control. Serializability was identified as the basis for preserving the consistency of a database in a centralized or distributed DBE. Issues regarding the conflicting operations from different transactions and their impact on the schedule that a DBE generates were discussed. We also discussed how locking and/or timestamping can be used to handle concurrent access to the database and to generate a serializable schedule. We identified deadlocks as a potential problem for locking-based approaches, but the discussion of deadlocks was delayed until Chapter 6. We discussed how locking and timestamping differ in the way they schedule transactions. We pointed out that the locking approach resulted in a nondeterministic commitment order for concurrent transactions, while timestamping provided for a deterministic order based on the age of each transaction.

The centralized concurrency control concepts then were extended to cover the issues in distributed DBEs. We explained the difference between a local schedule and a global schedule and outlined the rules for global serializability. Global serializability had two requirements—that local schedules be serializable and that all conflicting transactions commit in the same order at all the sites where they create a conflict.

We did not address the issue of atomicity when failures happen, because Chapter 8 discusses how atomicity is provided when there are failures in the system.

5.6 GLOSSARY

ACID Properties The properties that a transaction maintains in a traditional DBMS.

Aggressive TO Algorithm A version of the TO algorithm that tries to resolve conflicts as soon as possible.

ANSI American National Standards Institute.

ANSI Isolation Levels The set of concurrency control isolation levels that are part of the ANSI standard (four levels).

Atomicity A property of a transaction that requires all change or none of the changes of a transaction must be applied.

Backward Recovery The act of recovering a database to a consistent state in the past.

Blind Write The type of database write operation that does not require reading the value being changed.

Cascadeless Schedules A category of schedules that do not require cascade rollback when problems arise.

Cascading Rollback A transaction rollback operation that requires other transactions' rollback.

Centralized 2PL A version of 2PL algorithm that controls transactions from a centralized site.

Child Transaction A transaction that is activated as part of another transaction (usually a parent transaction).

Compensating Transaction A transaction that is run to compensate for the effect of a committed transaction.

Concurrency Control A database subsystem that is responsible for providing consistency of the database when there are multiple transactions running concurrently in the system.

Conflict A situation where two or more operations from different transactions are not allowed to be processed concurrently (typically a read and write or two writes).

Conflict Graph A graph that includes all conflicts across running transactions.

Conflict Matrix A matrix that shows compatible and incompatible transactions' operations.

Conflict Serializability A type of serializability that serializes conflicting operations from different transactions.

Conflict Serializable Schedules A schedule that is conflict serializable.

Conservative TO Algorithm A version of the TO algorithm that does not act on resolving conflicts immediately.

Consistency A requirement that every transaction must satisfy.

ConTracts A long transaction that consists of other long transactions.

Database A collection of data items where each item has a name and a value.

Database Consistency A requirement that guarantees database data items are consistent/correct.

DDL Lock A type of lock that is placed on database dictionary items.

Decision Support System A system that supports management of an organization in making business decisions.

Dirty Read A read operation where the changed value of a data item is read before the transaction that is changing it has been committed.

Distributed 2PL A version of 2PL that shares the control across all sites.

DML Lock A type of lock that is placed on behalf of a transaction's DML statements.

Durability A property of a transaction that requires writing its changed values to a persistent medium.

Exclusive Lock Mode A type of data item lock that cannot be shared.

Federated Databases An integration strategy for existing databases and database systems.

Forward Recovery An approach of recovering a database from an image in the past and reapplying each transaction's changes.

Global Transaction A transaction that works with data at more than one site in a distributed system.

Growing Phase The initial phase of 2PL when a transaction acquires locks.

History A total ordering of concurrent transactions' operations.

Index Page Locking An approach to dealing with the phantom issue, which utilizes locks on an index of a table.

Intention Locking A specific approach to locking, which requires a transaction to identify its intent on making changes to a lockable database granule.

Isolation A property of a transaction that forces the concurrency control to hide the changes of one transaction from the other transactions until it terminates.

Local Transaction A transaction that does not run at any site but the local site.

Lock Conversion The act of changing a lock from a finer granule to a coarser granule or vice versa.

Lock Downgrade The situation when a lock on a coarser granule is changed to a finer granule.

Lock Escalation The situation when a shared lock mode is changed to an exclusive lock mode.

Lock Matrix A matrix that outlines compatible and incompatible lock modes for a DBE system.

Lock Upgrade The situation when a lock on a finer granule is changed to a coarser granule.

Locking The act of isolating data items that are being accessed by transactions.

Logical Clock A counter that resembles the physical clock at a site.

Long-Lived Transaction A transaction that runs for a long time, usually minutes, hours, or even days.

Multidatabase An approach to integrating multiple existing databases and/or database systems.

Multiple-Granularity Locking A locking approach, which is usually used in conjunction with intention locking, that requires a transaction to lock not only the desired database granule but also the ancestors of it.

Multiversion Concurrency Control An approach to concurrency control that maintains multiple copies of each data item of the database.

One-Phase Locking A locking approach to concurrency control that allows transactions to lock and release items as they wish.

On-Line Analytical Processing A database environment in which reporting in support of management decisions is a prominent factor.

On-Line Transaction Processing A database environment in which many transactions run concurrently.

Optimistic Concurrency Control An approach to concurrency control that is designed to work well with low conflict environments.

Overwriting Uncommitted Data A type of isolation that allows one transaction to write what is being changed by another transaction.

Parallel Schedule An ordering of transactions in which one transaction starts before others finish.

Partial Commitment Order The commit order requirement that a conflicting operation from two different transactions puts on the system.

Pessimistic Concurrency Control An approach to concurrency control that is designed for high conflict environments.

Phantom Issue An issue that exists in relational database systems when the scope of rows that one transaction is working with is changed by other transactions' inserts or updates.

Point-in-Time Recovery A type of database recovery that restores a consistent database as of a point in time.

Predicate Locking An approach to dealing with the phantom issue in relational databases.

Pre-write The act of buffering the changes of one transaction in memory until the transaction is ready to commit/abort.

Primary Copy 2PL An approach to 2PL that uses a primary site for every data item of the database that is replicated.

Read Committed An isolation requirement that forces transactions to read only committed values.

Read Uncommitted Data An isolation requirement that allows transactions to read committed or uncommitted values.

Read-Only Transaction A transaction that does not make any changes to the database (a query).

Recoverable Schedules A schedule that can be recovered after a system or database failure of the system.

Saga A long transaction that contains other transactions nested in it.

Savepoint A consistent point in the life of Oracle transactions to which the database can be rolled back.

Schedule The total order of operations of concurrent transactions.

Sequential Schedule A schedule that only allows transactions to run serially (also known as serial schedule).

Serial Schedule A schedule that only allows transactions to run serially.

Serializability The requirement that forces the concurrency control of a database system to guarantee that any parallel schedule is equivalent to a serial schedule.

Serializable Schedule A parallel schedule that is equivalent to a serial schedule.

Shared Lock Mode A mode of database granule locking that can be used simultaneously by multiple transactions.

Short-Lived Transaction A transaction that runs for a very short period of time, usually subseconds.

Shrinking Phase The phase in 2PL when the transaction only releases the locks it has acquired.

SQL92 The Structured Query Language as per ANSI Standard 1992.

Strict Two-Phase Locking A specific type of 2PL where transactions have to hold the locked items until just before the commit.

System Lock A kind of lock that Oracle puts on objects in memory.

Timestamp Ordering An approach to concurrency control that uses the age of transactions as a way of enforcing serializability.

Total Commitment Order The commitment order of all concurrent transactions in the system.

Transaction A collection of reads, calculations, and writes of data items in a database.

Transaction Monitor A component or subsystem of a DBE that is responsible for managing the execution of a transaction.

Two-Phase Locking An approach to locking that disallows a transaction to request a lock after it has released a lock.

Unrepeatable Read An isolation level that does not guarantee the same value for what is read, if the same data item is accessed by the same transaction more than once.

View Serializability A serializability requirement that guarantees consistency of the database even though it does not produce a conflict serializable schedule.

View Serializable Schedule A schedule that is equivalent to a serial schedule.

REFERENCES

[Alsberg76] Alsberg, P., and Day, J., "A Principle for Resilient Sharing of Distributed Resources", In *Proceedings of 2nd International Conference on Software Engineering, San Francisco, CA*, pp. 562–570, 1976.

[Barker99] Barker, K., and Elmagarmid, A. "Transaction Management in Multidatabase Systems: Current Technologies and Formalisms," in *Heterogeneous and Autonomous Database Systems* (A. Elmagarmid et al., editors), Morgan Kaufmann, San Francisco, 1999, pp. 277–297.

[Bernstein80a] Bernstein, P., and Shipman, D., "The Correctness of Concurrency Control Mechanisms in a System for Distributed Databases (SDD-1)," *ACM Transactions on Database Systems*, Vol. 5, No. 1, pp. 52–68, March 1980.

[Bernstein80b] Bernstein, P., and Goodman, N., "Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems," in *Proceedings of the Sixth International Conference on Very Large Data Bases*, 1980.

[Bernstein87] Bernstein, P., Hadzilacos, V., and Goodman, N., *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading, M.A., 1987.

[Chrysanthis90] Chrysanthis, P. K., and Ramamritham K., "ACTA: A Framework for Specifying and Reasoning About Transaction Structure and Behavior," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 194–203, 1990.

[Elmagarmid90] Elmagarmid, A., Leu, Y., Litwin, W., and Rusinkiewics, M., "A Multidatabase Transaction Model for InterBase," in *Proceedings of the 16th International Conference on VLDB*, pp. 507–518, 1990.

[Eswaran76] Eswaran, K., Gray, J., Lorie, R., and Traiger, I., "The Notion of Consistency and Predicate Locks in a Database System," *Communications of the ACM,* pp. 624–633, 1976.

[Garcia-Molina87] Garcia-Molina, H., and Salem, K., "Sagas," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 249–259, 1987.

[Garcia-Molina91] Garcia-Molina, H., Salem, K., Gawlick, D., Klein, J., and Kleissner, K. "Modeling Long-Running Activities as Nested Sagas," *Database Engineering*, Vol. 14, No. 3, pp. 10–25, 1991.

[Gray81] Gray, J., "Transaction Concept: Virtues and Limitations," in *Proceedings of the 7th International Conference on Very Large Databases*, September 1981.

[Herman79] Herman, D., and Verjus, J., "An Algorithm for Maintaining Consistency of Multiple Copies," in *Proceedings of 1st International Conference on Distributed Computing Systems*, pp. 625–631, 1979.

[Kaiser92] Kaiser, G., and Pu, C., "Dynamic Restructuring of Transactions," in *Database Transaction Models for Advanced Applications* (A. Elmagarmid, editor), Morgan Kaufmann, San Francisco, 1992, pp. 265–295.

[Kung81] Kung, H., and Robinson, J., "On Optimistic Methods for Concurrency Control," *ACM Transactions on Database Systems*, Vol. 6, No. 2, pp. 213–226, June 1981.

[Leu91] Leu, Y., "Computing Multidatabase Applications Using Flex Transactions," *IEEE Data Engineering Bulletin* Vol. 14, No. 1, March 1991.

[Mohan86] Mohan, C., Lindsay, B., and Obermarck, R., "Transaction Management in the System R* Distributed Database Management System," *ACM Transaction Database System*, Vol. 11, No. 4, pp. 378–396, 1986.

[Papadimitriou82] Papadimitriou, C., and Kanellakis, P., "On Concurrency Control by Multiple Versions," in *Proceedings of the 1st ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, March 1982.

[Pu93] Pu, C., and Chen, S. W., "ACID Properties Need Fast Relief: Relaxing Consistency Using Epsilon Serializability," in *Proceedings of the Fifth International Workshop on High Performance Transaction Systems*, 1993.

[Reuter89] Reuter A., "Contracts: A Means for Extending Control Beyond Transaction Boundaries," in *Third International Workshop on High Performance Transaction Systems*, 1989.

[Sheth92] Sheth, A., Rusinkiewies, M., and Karabatis, G., "Using Polytransactions to Manage Interdependent Data," in *Database Transaction Models for Advanced Applications* (A. Elmagarmid, editor), Morgan Kaufmann, San Francisco, 1992, pp. 555–581.

[Silberschatz05] Silberschatz, A., Korth, H., and Sudarshan, S., *Database System Concepts*, McGraw-Hill, New York, 2005.

[Stonebreaker77] Stonebreaker, M., and Neuhold, E., "A Distributed Version of INGRES," in *Proceedings of the 2nd Berkeley Workshop on Distributed Data Management and Computer Networks*, Berkeley, CA, pp. 9–36, May 1977.

[Tandem87] The Tandem Database Group, "Non-stop SQL—A Distributed High Performance, High Availability Implementation of SQL," in *Proceedings of International Workshop on High Performance Transaction Systems*, September 1987.

[Tandem88] The Tandem Performance Group, "A Benchmark of Non-stop SQL on Debit Credit Transaction," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 337–341, June 1988.

[Veijalainen92] Veijalainen, J., Eliassen, F., and Holtkamp, B., "The S-Transaction Model," in *Database Transaction Models for Advanced Applications* (A. Elmagarmid, editor), Morgan Kaufmann, San Francisco, 1992, pp. 467–513.

[Weikum91] Weikum, G., "Principles and Realization Strategies of Multilevel Transaction Management," *ACM Transactions on Database Systems*, Vol. 16, No. 1, March 1991.

EXERCISES

Provide short (but complete) answers to the following questions.

5.1 Check the serializability of each one of the following schedules:

```
S1 = R1(X), R2(X), R3(X), W1(X), W3(X)
S2 = R2(X), R1(Y), W3(X), W2(X)
S3 = W1(X), R2(X), W3(X), W2(X)
```

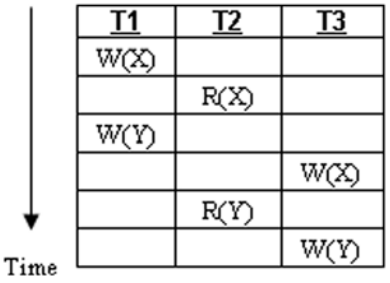


Figure 5.31 Concurrent transactions for Exercise 5.2.

```
S4 = R1(X), R3(X), R2(X), W2(Y), W3(X)
S5 = R3(X), W1(X), R1(X), W2(X)
```

Indicate all RW, WR, and WW conflicts and show all partial commitment orders forced by these conflicts. Indicate if the schedule is serializable. If it is, what is the total commitment order? If it is not, indicate the first operation that causes the problem.

5.2 Assume the following three transactions in Figure 5.31 and the order in which they work with data items X and Y. Show three schedules that are created by applying (A) no-locking, (B) 1PL, and (C) 2PL to these transactions. For each schedule in (A), (B), and (C), you should show all partial commitment orders (PCOs) caused by all RW, WR, and WW conflicts. Are these schedules serializable? If yes, what is the serialization order? If not, why?

5.3 Assume the three transactions in Figure 5.32 and the order in which they work with items X and Y. Answer the following for these three transactions, if we use (A) no-locking, (B) 1PL, and (C) 2PL for concurrency control. Show the schedules for (A), (B), and (C). Find out all partial commitment orders for RW, WR, and WW conflicts. If the schedule is serializable, what is the serialization order? If not, why? Are there any deadlocks?

5.4 Assume transactions T1, T2, T3, T4, and T5 generate schedules SA and SB as shown below. (A) Are the local SA and SB schedules serializable? If so, what are the equivalent local serial schedules? If not, why? (**Show all partial commitment**

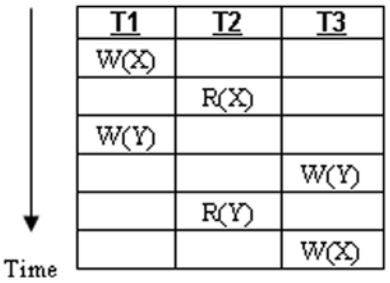


Figure 5.32 Concurrent transactions for Exercise 5.3.

orders.) (B) Are they globally serializable? If so, what is the equivalent global serial schedule? If not, why?

```
SA = R1(x1), W1(x1), R2(x2), R4(x1), W2(x2), W3(x2), R4(x2)
SB = R5(x3), R1(x3), R3(x4), W2(x4), R5(x4), R3(x3)
```

5.5 What is the condition for the optimistic concurrency control for transactions overlapping as (A) Ti completes its write phase before Tj starts its read phase, (B) Ti and Tj complete their validation phase at the same time, (C) Ti completes its validation phase before Tj completes its validation phase, (D) Ti completes its read phase after Tj completes its validation, and (E) both Ti and Tj complete their read phase exactly at the same time?

state, the algorithm is working on establishing a serializable order for a transaction. In the Queuing state, the algorithm has determined the order for a transaction and has entered the transaction in the queue. The protocols for achieving the goals of the PUA are similar to the two-phase commit (see Chapter 8). During the first phase of processing transaction "Ti," which is also called the "Voting" phase, the master of Ti asks all involved slaves to ready their queues for its transaction. Once all slaves have replied favorably, the master enters the second phase — the "Queuing" phase. To allow priority and fairness, the master can abort a transaction during the voting phase. However, once the queuing phase has started, the transaction will be queued and applied.

In response to the request to ready the queue, slaves act according to the state they are in:

- If a slave is in the idle state (i.e., not currently working on a transaction), then it votes "Ready."
- If a slave is in the active state (i.e., it has readied its queue for a conflicting transaction of higher priority), then it votes "Not Ready."
- If a slave has readied its queue for a conflicting transaction of lower priority, such as Tj, it will defer voting and tries to find out if it can change its vote for Ti. To do this, the slave sends a message to the master of Tj asking if its previous "Ready" vote can be changed to "Not Ready." If the master of Tj has not started its queuing phase, it will allow the vote change and aborts transaction Tj. Otherwise, the master of Tj will deny the vote change. The slave that deferred voting on Ti can then proceed accordingly.

It should be obvious that at any point in time the slave algorithm maintains two lists. The first list tracks all nonconflicting transactions that are pending to be queued. The second list tracks transactions that are deferred. Transactions are only vulnerable when they are in one of these lists. Once a transaction is queued, it is removed from the list it is in and it is guaranteed to be completed. To deal with failures, the two-phase commit is utilized between the master and corresponding LDMs. Before applying a transaction from the front of a queue, the LDM informs the master that it is ready to run the transaction. The master can then decide to continue with the transaction or abort it. If the decision is to abort, then this decision is communicated to all LDMs working on the corresponding transaction.

7.3 SUMMARY "Chapter 7"

In this chapter, we extended the concurrency control algorithms we discussed in Chapter 5 to include replication control. The issue with replication, as discussed in this chapter, is maintaining the mutual consistency of copies. Approaches were grouped into two categories. The first group, known as synchronous replication control, maintains identical database copies at all times. The second group, known as asynchronous replication control, allows the copies to show different values sometimes. We pointed out that, to detect a conflict, two conflicting transactions must be seen by at least one site. We exploited this concept for different algorithms that are proposed for replication control. We also discussed how voting was used to achieve the goal of replication control and concluded that different algorithms proposed for replication control, such

as consensus, majority, master–slave, posted updates, and token passing, were special cases of the generalized voting approach.

7.4 GLOSSARY

Asynchronous Replication An approach to replication control that allows copies of the database data items to show different values.

Broadcast The act of sending the same message to all sites in a network.

Circulating Token An approach to replication control that uses a token with a ticket to serialize transactions in a distributed database.

Consensus An approach to replication control that requires all sites to OK a transaction before it is run.

Daisy Chaining A distributed transaction execution application that requires a transaction to move from a site to a site that only allows it to be active at one site at any given point in time.

Fully Replicated Database A replication design that requires copying all tables of the database at all sites in the system.

Generalized Voting A comprehensive voting approach to replication control.

Majority Voting A voting approach to replication control that only requires a majority of sites to agree on applying a transaction.

Master–Slave A replication control strategy that assigns one site as the master and the others as the slaves.

Mutual Consistency The consistency requirement across copies of the same table and/or partition.

Partially Replicated Database A replication design that does not require copying all tables of the database at all sites in the system.

Posted Updates An approach to replication control that uses queues to order transactions for application to the copies of database tables.

Primary Copy The copy of a database that is designated at the primary—transactions are applied to this copy first.

Publisher The copy of a database that is designated at the publisher—transactions are applied to this copy first.

Read Quorum The number of votes required to read a data item in a replicated database.

Secondary Copy The copy of a database that is designated at the secondary—transactions are applied to this copy after they are applied to the primary.

Store and Forward Another terminology for primary–secondary replication control.

Subscriber The copy of a database that is designated at the subscriber—transactions are applied to this copy after they are applied to the publisher.

Symmetric Replication A replication control approach in which transactions are applied to the database copy at the site where they arrive.

Synchronous Replication An approach to replication control that does not allow copies of the database data items to show different values.

Transaction Acceptance Phase The phase in a replication control algorithm that decides whether or not to accept a transaction and orders its execution with other transactions.

Transaction Application Phase The phase in a replication control algorithm that applies a transaction that has already been serialized.

Weighted Voting A voting approach to replication control in which sites are given different numbers of votes to use during the transaction acceptance phase.

Write Quorum The number of votes required to write a data item in a replicated database.

REFERENCES

[Ellis77] Ellis, C., "A Robust Algorithm for Updating Duplicate Databases," in *Proceedings of the 2nd Berkeley Workshop on Distributed Data Management and Computer Networks*, pp. 146–158, 1977.

[Gifford79] Gifford, D., "Weighted Voting for Replicated Data," in *Proceedings of the 7th Symposium on Operating Systems Principles*, Pacific Grove, CA, ACM, New York, 1979, pp. 150–162.

[Nutt72] Nutt, G., "Evaluation Nets for Computer Analysis," in *Proceedings of FICC*, 1972, and also Ph.D. thesis, University of Washington, 1972.

[Petri62] Petri, C., "Kommunikation Mit Automation," Ph.D. thesis, University of Bonn, 1962.

[Rahimi79] Rahimi, S., and Franta, W., "The Effects of Topological, Replication, and Loading Factors on Concurrent Update Algorithm Performance: A Case Study," University of Minnesota, Computer Sciences Department Tech Report, 79-25, November 1979.

[Rahimi80] Rahimi, S., "A Posted Update Approach to Concurrency Control in Distributed Database Systems," Ph.D. thesis, University of Minnesota, 1980.

[Thomas79] Thomas, R., "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," *ACM Transactions on Database Systems*, Vol. 4, No. 2, pp 180–209, June 1979.

EXERCISES

Provide short (but complete) answers to the following questions.

7.1 Assume weighted voting is used for replication control of a five-site system, shown in Figure 7.7. Also assume that X as a data item is copied at all sites. How do you decide on the total number of votes, the number of votes assigned to each site, the read quorum (Vr(x)), and the write quorum (Vw(x)) that satisfies the following conditions:

(A) The number of votes assigned to each site is the absolute minimum? (**Note:** Zero is NOT accepted as the number of votes assigned to a site.)

(B) Site 1 can read if it decides to read X?

(C) X could be written if and only if Site 1 and at least two other sites vote to write?

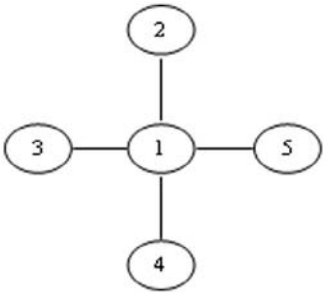


Figure 7.7 Network topology for Exercise 7.1.

7.2 There are two parts to the token passing replication control algorithm as mentioned in this chapter. The first part consists of ticketing transactions and the second part includes applying transactions. Draw the Petri net for each part of the algorithm. Do NOT worry about dealing with failures.

7.3 Assume the four-site system given in Figure 7.8 for the token passing replication control. Suppose, for fairness, when a site receives the token it can ticket a maximum of three transactions. A token is generated at Site 1 to start with. Answer the following questions for this setup for cycle 1 of the algorithm.

(A) What is the ticket number when the token leaves Site 1?

(B) What is the ticket number when the token comes back to Site 1?

(C) How many transactions and in what order are they serialized for the first cycle of the token (cycle starts at Site 1 and ends when the token arrives at Site 1 again)?

7.4 As depicted in Figure 7.9, during the second cycle, the communication line between Sites 3 and 4, when the token is on it, goes down. For this cycle, answer the following questions:

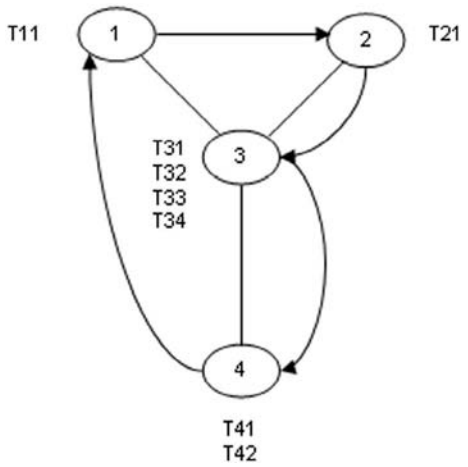


Figure 7.8 Network topology for Exercise 7.3.

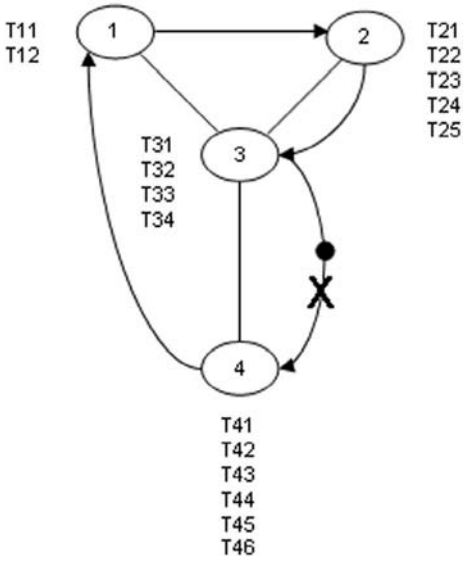


Figure 7.9 Network topology for Exercise 7.4.

(A) What is the ticket number when the token leaves Site 1?

(B) What is the ticket number when the token comes back to Site 1?

(C) How many transactions, and in what order, are serialized at the completion of cycle 1 and cycle 2?

7.5 At the beginning of cycle 3, the network looks like the one shown in Figure 7.10. Show the serialization order of all transactions at the completion of cycle 3.

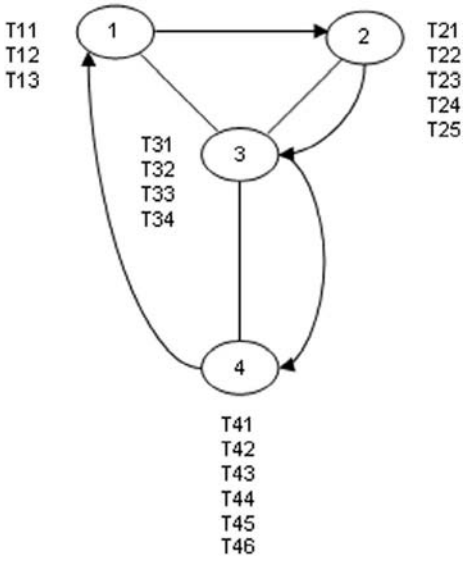


Figure 7.10 Network topology for Exercise 7.5.

must also have "0 ≺ = Vc ≺ = V" and "0 ≺ = Va ≺ = V." This simple idea makes it impossible for a partitioned network to commit and abort the same transaction at different partitions.

To make this work, the third phase of the 3PC is changed to count votes for abort or commit and to check to see which quorum, if any, is reached. This means that the sites are now required to attach their votes to the response they send to the coordinator when they receive the "Enter Prepared State" message. Under normal operations, the coordinator will collect the votes, which should be a quorum for commit, and then it will commit the transaction. Now suppose a failure partitions the network into two partitions when the system is the "Precommit" state. The coordinator will check the number of votes from the sites in its partition and it will decide if it can commit or abort the transaction. The slaves in the other partition elect a new coordinator and count their votes as well. With correct allocation of V, Vc, and Va, there is no way the two partitions can reach different quorums.

Let us apply this idea to the example discussed in Figure 8.18. Assume that each site is given one vote, making the total number of votes "V = 5." Let's also assume that "Va = 3" and "Vc = 3." This satisfies the requirement that "Va + Vc> V." In the partitioned network, if all computers in the right partition vote for commit, they have three votes to commit and have reached the commit quorum and will commit the transaction. Computers in the left partition, however, do not have enough votes to commit or abort and will be blocked from terminating the transaction. Although this is an undesirable situation, it is better than having the database become inconsistent. Obviously, the QBC protocol is blocking when a partition does not have a quorum to commit or abort the transaction.

8.9 SUMMARY "Chapter 8"

In this chapter, we analyzed the fault tolerance and recovery concepts for centralized and distributed databases. To guard against loss of database consistency, databases use a persistent medium, called a log, to store state information for transactions. After a failure, a system is brought back up; the log(s) are analyzed and depending on the state each transaction was in at the time of the failure, certain steps are taken. The overall approach for centralized and distributed systems is the same. What is different is that in a distributed system all sites must be recovered to the same consistent state. To achieve this, different distributed commit protocols are used. We analyzed one-phase, two-phase, three-phase, and quorum-based commit protocols for distributed database systems.

8.10 GLOSSARY

Active Transaction A transaction that is running at the time of the failure.

Before Commit State The state that a transaction enters just before it commits.

Checkpoint The process of writing the logged completed transactions to the stable storage.

Cold Backup A backup that is performed when there are no transactions running.

Commit Point A point in the life of a transaction when a decision has to be made to either commit or abort the transaction.

Commit Protocols A set of protocols that guarantee the consistency of the database even if failures happen.

Compete Backup A backup of the database that includes a complete snapshot of the database at the time of the backup.

Consistent Checkpointing The process that halts accepting new transactions, completes all active transactions, writes the proper log records to the log, and writes completed transactions to the stable storage.

Database Recovery The process of recreating a consistent snapshot of the database at a given point in time (usually as close as possible to the time of the failure).

Database Rollback The process of undoing the effects of some transactions to reinstantiate a consistent state of the database in the past.

Database Roll Forward The process of starting with a consistent snapshot of the database in the past and recreating a consistent state after that point by reapplying transactions.

Deferred Update An approach to database update that does not write the changes of a transaction to the database until the transaction is ready to commit.

Fuzzy Checkpointing The act of writing the list of active transactions to the log. It is usually used for recovery from power failure.

Global Transaction A transaction that works with data at more than one site in a distributed system.

Hard Failure A failure that causes loss of data in the database (the disk).

Hot Backup A backup that is performed while transactions are running in the system.

Immediate Update An approach to database update that writes the changes of a transaction to the database as soon as the transaction comes up with them.

Inactive Transactions A transaction that is not currently running because it is committed and/or aborted.

Incremental Backup A backup approach that only tracks changes since the last complete and/or incremental backup.

Local Transaction A transaction that does not run at any site but the local site.

Log Archival The process of writing the inactive portion of the log files to a secondary medium such as tape.

Log Buffers A prespecified amount of memory designated to the log records.

Nonvolatile Storage A storage medium that can withstand power failures.

One-Phase Commit (1PC) A distributed commit approach in which the decision to commit and/or abort is communicated in one broadcast.

Point-In-Time (PIT) Recovery The process of recovering a consistent snapshot of the database to a point in time.

Quiescent Point A point in time when no transactions are running in the system.

Quorum-Based Commit (QBC) A modified version of the 3PC that uses voting to deal with network failures during the third phase.

Recovery Protocol The protocol that a failed system must follow after it has been repaired.

Recovery to Current The process of recovering the database to the present time (usually the time of the failure or as close to it as possible).

Redo The process of reapplying the changes of a transaction.

Redo Scripts A set of redo operations for some transactions.

Soft Failure A failure that does not cause loss of data on disk—usually caused by some form of "unclean" shutdown, such as a power failure, operating system misbehavior, or DBMS bug.

Stable Storage A storage medium that does not lose its contents even when disk failures happen.

System Crash A failure caused by loss of power to the system; a failure that does not cause loss of data on disk.

Termination Protocol The process that a site must follow when the site notices that its counterpart has failed.

Three-Phase Commit (3PC) A distributed commit approach in which the decision to commit and/or abort is communicated in three broadcasts.

Transaction Log A record of all changes that transactions have made to the database.

Transaction Recovery The process of reapplying the changes of a transaction.

Two-Phase Commit (2PC) A distributed commit approach in which the decision to commit and/or abort is communicated in two broadcasts.

Undo The process of getting rid of the changes of a transaction from the database.

Undo Scripts A set of undo operations for a set of transactions.

Volatile Storage A storage medium that loses its contents when power to it is lost.

REFERENCES

[Astrahan76] Astrahan, M., Blasgen, M., Chamberlin, D., Eswaran, K., Gray J., Griffiths, P., King, W., Lorie, R., McJones, P., Mehl, J., Putzolu, G., Traiger, I., Wade, B., and Watson V., "System R: A Relational Approach to Database Management," *ACM Transactions on Database Systems*, Vol. 1, No. 2, pp. 97–137, June 1976.

[Attaluri02] Attaluri, G., and Salem, K., "The Presumed-Either Two-Phase Commit Protocol," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 14, No. 5, pp. 1190–1196, September 2002.

[Bernstein87] Bernstein, P., Hadzilacos, V., and Goodman N., *Concurrency Control and Recovery in Database Systems*, Addison-Wesley Longman Publishing Company, Boston, MA, 1987.

[Gray78] Gray, J., Bayer, R., Graham, R., and Seegmuller, G., "Notes on Data Base Operating Systems," in *Operating Systems: An Advanced Course*, Notes in Computer Science, Vol. 60, Springer-Verlag, New York, 1978, pp. 393–481.

[Gray81] Gray, J., McJones, P., Blasgen, M., Lindsay, B., Lorie, R., Price, T., Putzolu, F., and Traiger, I., "The Recovery Manager of the System R Database Manager," *ACM Computing Survey*, Vol. 13, No. 2, pp. 223–242, June 1981.

[Gray87] Gray, J., "Why Do Computers Stop and What Can Be Done About It?" Technical Report 85.7, Tandem Corporation, 1985. Also in "Tutorial Notes, Canadian Information Processing Society," Edmonton '87 Conference, Edmonton, Canada, November 1987.

[Lampson76] Lampson, B., and Sturgis, H., "Crash Recovery in Distributed Data Storage System," Tech Report, Xerox Palo Alto Research Center, Palo Alto, CA, 1976.

[Lampson93] Lampson, B., and Lomet, D., "A New Presumed Commit Optimization for Two Phase Commit," in *Proceedings of the 10th VLDB Conference*, Dublin, Ireland, pp. 630–640, 1993.

[Lindsay79] Lindsay, B., "Notes on Distributed Databases," IBM Technical Report RJ2517, San Jose, CA, July 1979—also Tandem Technical Report 81.3, June 1981.

[Mohan85] Mohan, C., and Lindsay, B., "Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions," *ACM SIGOPS Operating Systems Review*, Vol. 19, No. 2, pp. 76–88, April 1985.

[Mourad85] Mourad, S., and Andres, D., "The Reliability of the IBM/XA Operating System," in *Proceedings of the 15th Annual International Symposium on Fault-Tolerant Computing Systems*, pp. 93–98, 1985.

[Özsu99] Özsu99, M., and Valduriez, P., *Principles of Distributed Database Systems*, Prentice Hall, Englewood Cliffs, NJ, 1999.

[Petri62] Petri, C., "Kommunikation Mit Automation," Ph.D. thesis, University of Bonn, 1962.

[Severance76] Severance, D., and Lohman, G., "Differential Files: Their Application to the Maintenance of Large Databases," *ACM Transactions on Database Systems*, Vol. 1, No. 3, pp. 256–261, September 1976.

[Skeen81] Skeen, D., "Non-blocking Commit Protocols," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 133–142, 1981.

[Skeen82] Skeen, D., "A Quorum-Based Commit Protocol," Tech Report TR82–483, Cornell University, Ithaca, NY, 1982.

[Skeen83] Skeen, D., and Stonebraker, M., "A Formal Model of Crash Recovery in a Distributed System," *IEEE Transactions on Software Engineering*, Vol. 9, No. 3, pp. 219–228, May 1983.

EXERCISES

Provide short (but complete) answers to the following questions.

8.1 A quorum-based commit protocol is implemented to deal with network partitioning of a four-site system. Sites 1 and 3 have the same importance. Sites 2 and 4 are more important than Sites 1 and 3. Site 2 is more important than Site 4. How do you decide on the **smallest** number of votes for V (total votes), V1, V2, V3, V4, Vc, and Va if we want the partitioned system to abort if and only if Sites 2 and 4 are in the same partition (**but** cannot abort if Sites 2 and 4 are in different partitions)?

8.2 Assume the transactions shown in Figure 8.19, the time of checkpoints, and the time of failure. Suppose the log contains the "Before Commit" records for T5 and T6 but does not contain the "Before Commit" record for T7.

(A) Assume that we utilize a deferred update approach. Explain the recovery steps from a soft crash (power failure). Explain the recovery steps from a disk failure (hard crash).

(B) Now assume that the update approach is immediate updates. Explain the recovery steps from a soft crash (power failure). Explain the recovery steps from a disk failure (hard crash).

(C) Assume that during the recovery from a power failure in the deferred update case, a disk crash happens. Explain the recovery steps from this failure.

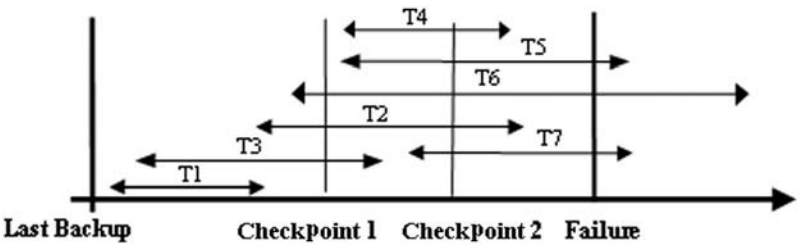


Figure 8.19 Transactions for Exercise 8.2.

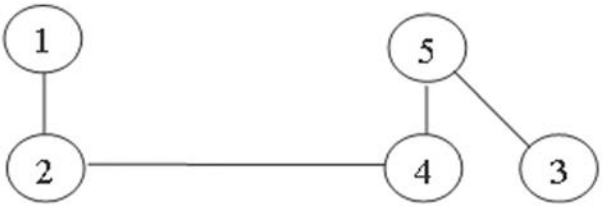


Figure 8.20 Topology for computers in Exercise 8.3.

8.3 Assume every site in Figure 8.20 has the same weight for voting in a quorum-based commit protocol. What are "Vi (for i = 1 to 5)," V, Vc, and Va for the following cases?

(A) We do NOT want the system to commit or abort any transaction when the link between 2 and 4 goes down.

(B) We want each partition to commit but no partition is able to abort when the link between 2 and 4 goes down.

(C) We want each partition to abort but no partition is able to commit when the link between 2 and 4 goes down.

(D) If Site 4 goes down, can the partitions in the partitioned system achieve a different quorum if each site is given one vote? Explain your answer.

The application server code carefully validates any user input to ensure there are no buffer-overruns or SQL injection attacks or other types of accidental or malicious input. Application server objects are retrieved and stored to the third tier distributed network of databases. The communication between the application server and the databases, and between the databases themselves, is also secured with TLS. The application server authenticates itself to the DDBE, typically with a user ID and password that is protected using the application server operating system host security mechanisms, such as a protected registry or keychain.

The database tier uses its SQL security model to ensure that requests are authorized. Each database, if the data is sensitive, might also be encrypted so that theft of hard disks would not result in data compromise. The database backup tapes must also be encrypted so that their storage or disposal will not result in a compromise.

The database tier and application tier execute a strict audit policy and log all significant access attempts, successes, and failures. These audit logs are also protected from modification by leveraging operating system facilities or special purpose, external, one-way, write-only logging servers. Firewalls and intrusion detection systems can be strategically placed across the network to block unwanted traffic and to detect attack attempts.

The security deployment must be validated, usually via a security audit, and all software must be kept up-to-date with patches at all levels. A penetration (PEN) test would further assure the implementation by having an experienced security tester (known as a white hat) try to compromise the system before an attacker (known as a black hat) will. Finally, an organization must have a written set of security processes that implement security policies for system maintenance, testing, incident response, and user and administrator authentication and authorization.

9.6 SUMMARY "Chapter 9 "

Securing computing systems is a very broad topic. The issues we must consider come from several different areas: physical security, host and operating system security, network security, and application security. In order to address the issues in these areas, we must perform many tasks, such as ensuring good coding practices, enforcing good policies and procedures for auditing, and assuring, detecting, and reacting to attacks. Most of the technologies for securing distributed databases focus on the data itself and the communication paths. These protocols are in turn based on cryptographic building blocks and specific implementation algorithms for these blocks. In this chapter, we explored the basic building blocks and associated algorithms. We then looked at some basic protocols for securing communications and data, examined several database-specific security vulnerabilities, examined some broader architectural issues, and looked at a typical deployment. As the amount of online data and the number of people granted access to it (worldwide) continue to increase at a staggering rate, security issues and technologies will only continue to grow in importance.

9.7 GLOSSARY

Advanced Encryption Standard (AES) A conventional symmetric block cipher standard, selected by NIST to replace DES in 2002, based on the Rijndael cipher submission that uses a 128-bit block size with 128-, 192-, or 256-bit key lengths.

Arbitrated Digital Signatures A family of digital signature protocols that leverage a trusting third party and can help limit the risk of stolen private keys.

Authentication Any technique used to validate the identity of a source or destination of messages or data.

Authorization Any technique used to determine whether a particular user, component, or subsystem has sufficient access permissions to perform a desired operation—this is usually checked after the identity has been authenticated.

Block Cipher A cipher that encrypts and decrypts a group of bits at a time and that does not save state.

Brute Force Attack An attempt to break encryption by trying all the keys.

Certificate Authority A company that verifies the identities associated with public keys and then, usually for a fee, signs those keys when they are embedded in digital certificates, to be subsequently verified by the corresponding certificate authority public keys present in browsers and operating systems.

Certificate Revocation List (CRL) A list of invalid certificate serial numbers maintained by each certificate authority.

Cipher A form of cryptography that maps symbols (i.e., bytes) to other symbols.

Cipher Block Chaining (CBC) A cryptographic mode where a given block encryption/decryption depends not only on the current block input but also on the previous block output, forming a dependency chain.

Ciphertext The message or data after encryption prior to decryption.

Code A form of cryptography that maps words to other words.

Confidentiality Any technique that ensures that data is kept private from eavesdroppers, usually by using encryption.

Conventional Cryptography A cipher that uses a common shared key known only to the sender and receiver.

Cryptanalysis The science of breaking secrets.

Cryptography The science of creating secrets.

Data Encryption Standard (DES) A conventional symmetric block cipher standard selected by NIST in 1976 based on an IBM submission, which uses a 64-bit block size and a 56-bit key.

Decryption Any technique or operation that recreates plaintext, or the message/data to be communicated, from ciphertext.

Diffie–Hellman A public key agreement algorithm based on the difficulty of computing discrete logarithms in a finite field.

Digital Certificate A public key and other identifying information, usually in X.509 format, signed with a certificate authority's private key after the certificate authority verifies the public key identity.

Digital Signature A cryptographic technique that uses a source private key to sign data such that a destination can use the source public key to authenticate the source with the strength of nonrepudiation.

Electronic Code Book (ECB) A cryptographic mode where a given block encryption/decryption depends only on the current block input.

Encrypted File System (EFS) A file system where the disk blocks are encrypted, preventing data compromise if the disk is physically stolen.

Encryption Any technique or operation that creates ciphertext, which is confidential in the face of eavesdroppers, from plaintext, which is the underlying message/data.

Entropy A measure of the true information content, in bits, of a password that takes into account the probability of occurrence of each possible password symbol.

Firewall A network security component that controls access to and from a computing system that can screen out traffic based on network address, port, protocol, or contents.

Hash Another term for a message digest operation.

Hashed Message Authentication Code A framework, based on a specific message digest algorithm, for creating a message authentication code.

Initialization Vector (IV) A nonsecret, never reused, bit string (usually the same length as a cipher block) that starts the chain of encryption and decryption for modes such as cipher block chaining (CBC).

Integrity A property in security, anything that ensures that a message/set of data is not modified by accident or malicious intent.

Intrusion Detection System (IDS) A network security component that detects malicious network packet signatures indicating a potential virus, worm, or other attack.

Key A bit string that controls a cipher's mapping of plaintext to ciphertext for encryption and ciphertext to plaintext for decryption.

Lightweight Directory Access Protocol An architecture for a read-mostly database, which can contain authentication and authorization data.

Man-in-the-Middle Attack An attack where an active attacker can read encrypted messages/data only intended for two parties in a manner such that the two parties are unaware of the compromise.

MD5 A message digest algorithm with a 128-bit hash size.

Meet-in-the-Middle Attack An attack against two successive encryption stages with two distinct keys and a known plaintext/ciphertext pair, where encryption of the plaintext with one key is matched with ciphertext decryption of another key. A match signifies that both keys are now known. This attack reduces Double DES to Single DES strength.

Message Authentication Code A message digest that also uses a conventional key as part of the fingerprint computation.

Message Digest A small, fixed sized, digital fingerprint (or hash) for data of arbitrary input size using a one-way algorithm that makes the computation of the message digest fast, but the creation of another input that matches the fingerprint computationally infeasible.

Mode An algorithm for breaking up plaintext that is larger than the cipher block size in a cryptographically secure manner, usually by making the current block encryption dependent on previous block encryptions.

Nonrepudiation The inability to deny being the source of a message or data.

One Time Pad (OTP) A theoretically perfectly secure cipher that uses a perfectly random key that is as long as the plaintext, which the sender XORs to the plaintext to produce the ciphertext. The receiver, who has the same key, XORs the key to the ciphertext to recreate the plaintext.

Padding The addition of bits to fill out a full block size prior to encryption in a cryptographically secure manner.

Plaintext The message/data prior to encryption or after decryption.

Private Key The key from the public key cryptography key pair used for decryption and signing that must be kept secret.

Public Key The key from the public key cryptography key pair used for encryption and signature verification that can be revealed to the public.

Public Key Cryptography A cipher that uses a public/private key pair for encryption or digital signature. For encryption, a source uses the destination public key to encrypt data that the destination can decrypt using its private key.

RC4 A stream cipher, commonly used in TLS/SSL and wireless network encryption, that models a one time pad using a seed as a shared key, which is the input to a pseudorandom number generator.

Rijndael The name of the cipher (now called AES) that won the NIST competition for DES replacement in 2002.

RSA A public key cryptosystem that can be used for encryption and digital signature based on the difficulty of factoring large numbers into component primes.

Secure Shell (SSH) A protocol for secure remote telnet that can also be used for building secure communication channels between two endpoints.

SHA-1, SHA-256 A message digest algorithm with a 160-bit hash size.

Single Sign On (SSO) The concept of consolidating authentication and authorization information in a centrally managed and accessed location.

SQL Injection An attack against a database that typically leverages unvalidated web-form input, which results in malicious SQL statements when these input values are carefully crafted and subsequently concatenated with dynamic SQL statements.

Steganography The science of hiding secrets.

Stream Cipher A cipher that encrypts or decrypts 1–8 bits at a time dependent on input data and previous state.

TLS/SSL A protocol for encrypting communications while providing data integrity, which also authenticates servers to clients (and optionally clients to servers).

Triple DES A cipher composed of three sequential DES ciphers, which offers twice the security as DES and uses a 64-bit block size and 112- or 168-bit keys. Also known as 3DES and DESede.

Virtual Private Network (VPN) A protocol that enables a public, insecure network to be used as if it were private with the use of encryption for confidentiality and digital signature for authentication.

REFERENCES

[Anderson08] Anderson, R. J., *Security Engineering: A Guide to Building Dependable Distributed Systems*, Wiley, Hoboken, NJ, 2008.

[Daemen02] Daemen, J., and Rijmen, V., *The Design of Rijndael: AES—The Advanced Encryption Standard*, Springer, Berlin, 2002.

[Denning83] Denning, D. E., and Schlorer, J., "Inference Controls for Statistical Data Bases," *IEEE Computer* 16, 7 July 1983, 69–82.

[Diffie76] Diffie, W., and Hellman, M., "Multiuser Cryptographic Techniques," in *Proceedings of the AFIPS National Computer Conference*, pp. 109–112, 1976.

[Ferguson03] Ferguson, N., and Schneier, B., *Practical Cryptography*, Wiley, Hoboken, NJ, 2003.

[Howard05] Howard, M., LeBlanc, D., and Viega, J., *19 Deadly Sins of Software Security*, McGraw-Hill Osborne Media, New York, 2005.

[Natan05] Natan, R. B., *Implementing Database Security and Auditing: Includes Examples for Oracle, SQL Server, DB2 UDB, Sybase*, Digital Press, Clifton, NJ, 2005.

[Rivest78] Rivest, R., Shamir, A., and Adleman, L., "A Method for Obtaining Digital Signatures and Public Key Cryptosystems," *Communications of the ACM*, Vol. 21, pp. 120–126, 1978.

[Stallings05] Stallings, W., *Cryptography and Network Security*, 4th edition, Prentice Hall, Englewood Cliffs, NJ, 2005.

EXERCISES

Provide short (but complete) answers to the following questions.

9.1 Suppose Alice and Bob each have a public key pair. Alice sends a signed and encrypted message to Bob, who processes the message. Show the order in which the four keys are used in this scenario.

9.2 Describe at least three reasons why one time pads (OTPs) are impractical.

9.3 Suppose we wanted to go into the certificate authority (CA) business to compete with Verisign. What would we need to do?

9.4 What two functions are performed by the public key cryptographic technology in SSL/ TLS?

9.5 Suppose Alice wants to send a conventionally encrypted message to Bob but they have never met before or agreed on any specifics of the message exchange. Name five things Bob must be told before he can decrypt and interpret the message from Alice.

9.6 Suppose we had a business with an online storefront and our private key corresponding to our SSL/TLS certificate has just been stolen. What actions must we take, and what actions must our customers take?

9.7 Using the same scenario as in Figure 9.5, what would happen if an attacker entered the following values for the web-form fields?

```
InputName = '; DELETE FROM CustomerLogin WHERE ''='

InputPass = ' OR ''='
```

9.8 What is the difference between database authentication and authorization?

9.9 What is the difference between a firewall (FW) and an intrusion detection system (IDS)?

9.10 What role does auditing play in securing databases?