Table of Contents for Transaction Processing

1

Introduction

Six thousand years ago,

the Sumerians invented writing

for transaction processing

1.1 Historical Perspective

Six thousand years ago, the Sumerians invented writing for transaction processing. The earliest known writing is found on clay tablets recording the royal inventory of taxes, land, grain, cattle, slaves, and gold; scribes evidently kept records of each transaction. This early system had the key aspects of a transaction processing system (see Figure 1.1):

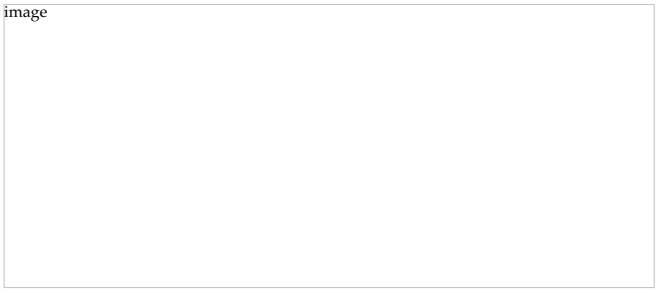


FIGURE 1.1 **The basic abstraction of transaction processing systems**. The real state is represented by an abstraction, called the *database*, and the transformation of the real state is mirrored by the execution of a program, called a *transaction*, that transforms the database

Database. An abstract system state, represented as marks on clay tablets, was maintained. Today, we would call this the *database*.

Transactions. Scribes recorded state changes with new records (clay tablets) in the database. Today, we would call these state changes *transactions*.

The Sumerians' approach allowed the scribes to easily ask questions about the current and past state, while providing a historical record of how the system got to the present state.

The technology of clay-based transaction processing systems evolved over several thousand years through papyrus, parchment, and then paper. For over a thousand years, paper, ink, and ledgers were the technology for transaction processing. The most recent innovation began late in the 1800s when Herman Hollerith built a punched-card computer system to record and report the 1890 United States census. During the first half of the twentieth century, the need for transaction processing fueled the evolution and growth of punched-card equipment. These early computers were used primarily for inventory control and accounting. In effect, they replaced clay tablets with paper tablets (cards); their virtue was that the systems could search and update about one "tablet" (card) per second.

The second half of the twentieth century saw two main developments in transaction processing: *batch* transaction processing based on magnetic storage (tape and disc), followed by *online* transaction processing based on electronic storage and computer networks. These two developments were largely responsible for growth in the computer industry, and transaction processing applications accounted for the majority of computer systems revenues. Today, the primary use of general-purpose computers is still transaction processing. Typical applications and examples include the following:

Communications. Setting up and billing for telephone calls, electronic mail, and so on.

Finance. Banking, stock trading, point of sale, and so on.

Travel. Reservations and billing for airlines, hotels, cars, trains, and so on.

Manufacturing. Order entry, job and inventory planning and scheduling, accounting, and so on.

Process control. Control of factories, warehouses, steel, paper, and chemical plants, and so on.

Consider the example of a telephone call. Each time you make a phone call, there is a call setup transaction that allocates some resources to your conversation; the call teardown is a second transaction, freeing those resources. The call setup increasingly involves complex algorithms to find the callee (800 numbers could be anywhere in the world) and to decide who is to be billed (800 and 900 numbers have complex billing). The system must deal with features like call forwarding, call waiting, and voice mail. After the call teardown, billing may involve many phone companies (e.g., direct dial from San Francisco to Madagascar involves several phone companies).

As another example, computer integrated manufacturing (CIM) is a key technique for improving industrial productivity and efficiency. Just-in-time inventory control, automated warehouses, and robotic assembly lines each require a reliable data storage system to represent the factory state. In addition, computers control and monitor the flow of goods through the factory.

As with most modern enterprises, these two applications—telephony and manufacturing—require vast quantities of software. In the past, such systems were often implemented using specialized computers and ad hoc techniques. This meant that the applications had to be reprogrammed for each new generation of computers. However, the development effort for these applications has become so huge that the investment must be preserved for several hardware generations. As a result, standard software is now used to ensure that the application will work on the next generation of computer systems. Project risks and costs are also reduced by using high-level tools to improve programmer productivity. Consequently, system control functions are being implemented using standard transaction processing systems, standard operating systems, standard database systems, and general-purpose hardware.

1.2 What Is a Transaction Processing System?

This chapter views a transaction processing system from many different perspectives: a user perspective, a programmer perspective, an administrator perspective, and a TP system implementor perspective. Because each views the system quite differently, it is difficult to give a single definition of what a transaction processing system is and what it does. If forced to do so, however, most will agree to the following statement:

> *A transaction processing system (TP system) provides tools to ease or automate application programming, execution, and administration. Transaction processing applications typically support a network of devices that submit queries and updates to the application. Based on these inputs, the application maintains a database representing some real-world state. Application responses and outputs typically drive real-world actuators and transducers that alter or control the state. The applications, database, and network tend to evolve over several decades. Increasingly, the systems are geographically distributed, heterogeneous (they involve equipment and software from many different vendors), continuously available (there is no scheduled down-time), and have stringent response time requirements.*

The term transaction processing system is generally used to mean a *complete* system. A TP system includes application generators, operations tools, one or more database systems, utilities, and, of course, networking and operating system software. Historically, TP system meant *Tele-Processing system* and denoted a program supporting a variety of terminal types and networking protocols. Some current transaction processing systems evolved from teleprocessing systems, further complicating the terminology. As used here, the term *TP system* has the connotation of transaction processing.

A TP system is a big thing; it includes application generators, networks, databases, and applications. Within the TP system, there is a core collection of services, called the *TP monitor*, that manage and coordinate the flow of transactions through the system.

All these definitions use the term *transaction*. What exactly is a transaction? It has the following various meanings:

(1) The request or input message that started the operation. *transaction request/reply*

(2) All effects of the execution of the operation. *transaction*

(3) The program(s) that execute(s) the operation. *transaction program*

These ambiguities stem from the various perspectives of people involved in the transaction. The end user sees only the request and reply and, consequently, thinks in those terms. The operator and the auditor primarily see the request execution, so they take that view. The system administrator largely deals with the naming, security, and the transaction programs; he therefore often thinks of the transaction as the program source rather than the program execution.

This book adopts the second definition: *A transaction is a collection of operations on the physical and abstract application state.* The other two transaction concepts are called the transaction request/reply and the transaction program.

Transaction processing systems pioneered many concepts in distributed computing and fault-tolerant computing. They introduced distributed data for reliability, availability, and performance; they developed fault-tolerant storage and fault-tolerant processes for availability; and they developed the client-server model and remote procedure call for distributed computation. Most important, they introduced the transaction ACID properties—atomicity, consistency, isolation, and durability—that have emerged as the unifying concepts for distributed computation.

This book contains considerably more information about the ACID properties. For now, however, a transaction can be considered a collection of actions with the following properties:

Atomicity. A transaction's changes to the state are atomic: either all happen or none happen. These changes include database changes, messages, and actions on transducers.

Consistency. A transaction is a correct transformation of the state. The actions taken as a group do not violate any of the integrity constraints associated with the state. This requires that the transaction be a correct program.

Isolation. Even though transactions execute concurrently, it appears to each transaction, *T*, that others executed either before *T* or after *T*, but not both.

Durability. Once a transaction completes successfully (commits), its changes to the state survive failures.

As an example, a banking debit transaction is atomic if it both dispenses money and updates your account. It is consistent if the money dispensed is the same as the debit to the account. It is isolated if the transaction program can be unaware of other programs reading and writing your account concurrently (for example, your spouse making a concurrent deposit). And it is durable if, once the transaction is complete, the account balance is sure to reflect the withdrawal.[1]

It may be hard to believe that such a simple idea could be very important, but that is the point. Transactions specify simple failure semantics for a computation. It is up to the system implementors to provide these properties to the application programmer. The application programmer, in turn, provides these properties to his clients.

The application program declares the start of a new transaction by invoking Begin_Work(). Thereafter, all operations performed by the program will be part of this transaction. Also, all operations performed by other programs in service of the application program will be part of the transaction. The program declares the transaction to be a complete and correct transformation by invoking Commit_Work(). Once the transaction successfully commits, the transaction's effects are durable. If something goes wrong during the transaction, the application can undo all the operations by invoking Rollback_Work(). If there is a failure during the transaction's execution, the system can unilaterally cause the transaction to be rolled back. Begin-Commit or Begin-Rollback, then, are used to bracket the ACID transformations.

This is a simple and modular way to build distributed applications. Each module is a transaction or sub-transaction. If all goes well, the transaction eventually commits and all the modules move to a new durable state. If anything goes wrong, all the modules of the transaction are automatically reset to their state as of the start of the transaction. Since the commit and reset logic are automatic, it is easy to build modules with very simple failure semantics.

Many of the techniques used to achieve the ACID properties have direct analogies in human systems. In retrospect, we have just abstracted them into computer terms. The notions of atomicity and durability are explicit in contract law, in which a notary or escrow officer coordinates a business transaction. Even more explicit is the Christian wedding ceremony, in which two people agree to a marriage. The marriage commit protocol goes as follows: The minister asks each partner, "Do you agree to marry this person?" If both say "yes," then the minister declares the marriage committed; if either fails to respond, or says "no," then the marriage is off (neither is married). Transaction processing systems use a protocol directly analogous to this to achieve all-or-nothing atomic agreement among several independent programs or computers. Translated into computer terms, this idea is called the *two-phase commit protocol*: it has a voting phase, in which all participants prepare to commit, followed by a second commit phase, in which they execute the commit. A distinguished process acts much as the minister, coordinating the commit. This protocol allows all participants in a distributed computation to either agree to change state or agree to stay in the old state.

The transaction concept is the computer equivalent to contract law. Imagine a society without contract law. That is what a computer system would be like without transactions. If nothing ever goes wrong, contracts are just overhead. But if something doesn't quite work, the contract specifies how to clean up the situation.

There are many examples of systems that tried and failed to implement fault-tolerant or distributed computations using ad hoc techniques rather than a transaction concept. Subsequently, some of these systems were successfully implemented using transaction techniques. After the fact, the implementors confessed that they simply hit a complexity barrier and could not debug the ad hoc system without a simple unifying concept to deal

with exceptions [Borr 1986; Hagmann 1987]. Perhaps even more surprising, the subsequent transaction-oriented systems had better performance than the ad hoc incorrect systems, because transaction logging tends to convert many small messages and random disk inputs and outputs (I/Os) into a few larger messages and sequential disk I/Os.

This book focuses on the definition and implementation of the ACID properties. First, let's look at transaction processing systems from the perspectives of various people who use or operate such systems: users, system administrators, system operators, application designers, and application implementors. Each has a different view of the system. Later chapters take the system implementor's perspective—the person who has to implement all these other views.

A sample application system is needed to make the various views concrete. Funds transfer is the standard example application used for transaction processing systems because it deals with a simple, abstract, and valuable commodity. It is also one of the major application areas of transaction processing systems. For the sake of variety, let's consider a forms-oriented electronic mail application from the various perspectives.

1.2.1 The End User's View of a Transaction Processing System

The end user's view of the mail system is that there are mailboxes associated with people, and there are messages (see Figure 1.2). The user first identifies himself to the system and presents a password or some other item that authenticates him to the system. When this transaction completes, the user is presented with a list of incoming messages. The user can read a message, reply to it, delete it, send a new message, or cancel a message he has sent. Each of these operations is an ACID transaction. The read operation can see the whole message (no part of the message will be missing), while the delete operation removes the whole message. The interactions between send and cancel is the most instructive aspect of this example. If the system fails while the user is composing a message and before the user issuing a send, then the message is not sent and the user's input is lost. After the user successfully issues the send, the message is delivered. Thus, the send is an ACID transaction. Two more transactions are involved: The message is delivered to the receiver's mailbox, and the message is read by the receiver. These are also ACID units. The sender may cancel (erase) an unread message, but the cancel will have no effect if the message has already been read by the receiver. The cancel is another ACID transaction; it is called a *compensating* transaction, since it is trying to compensate for the effects of a previously committed transaction.

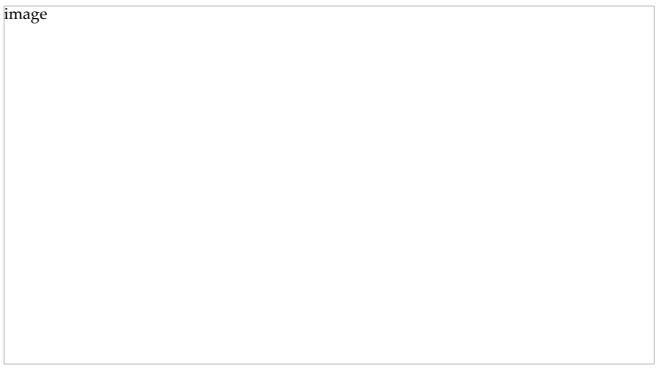


FIGURE 1.2 End user's view of a transaction processing system as a set of operations on abstract objects In this case, the objects are mail messages and mailboxes. At the left, the diagram shows the menu hierarchy of operations that the user sees. The objects on the right are the transactional objects managed by the system. The user expects each operation to be atomic, consistent, isolated, and durable (ACID). The operations are presented to the user as input and output fields containing text, sounds, or images. To generalize this model, the end user views the system as a pre-programmed device with well-defined *operations on objects*. The object states are durable (persistent, stable), and the operations are atomic (all-or-nothing), consistent (transforms of high-level abstractions such as mail messages and mailboxes are complete), isolated (concurrent updates are applied sequentially), and durable (messages are not lost).

The operation of composing a mail message may involve updating many database records representing the message, and it may require inquiries to remote name servers tocheck the correctness of the mail address; all this work is packaged within the one ACID unit. If anything goes wrong during the message composition, all the operation's updates are reversed, and the message is discarded; if all goes well, all the operation's updates are made durable, and the message is accepted for delivery. Similarly, delivery of the message to the destination may encounter many problems. In such cases, the transaction is aborted, and a new delivery transaction is started. Eventually, the message is delivered to (inserted in) the destination mailbox. After that, it can be read by the recipient and replied to or deleted. Of course, to make the message durable, it is probably replicated in two or more places so that no single fault can damage it; that, again, is transparent to the end user, who thinks in terms of composing, sending, reading, and canceling messages. For each of these operations, the electronic mail application provides the end user with ACID operations on durable objects.

In this example, operations are shown to humans as a forms-oriented interface with text, voice, and button inputs. Output forms contain text, sounds, and images. In many other applications, the *user* is not a person, but rather a physical device such as a telephone switch, an airplane flap, or a warehouse robot. In these cases, the inputs are the readings of position and tactile sensors, while the outputs are commands to actuators. The sensors present the transaction processing system with input messages that invoke a transaction, and the transducers act on the transaction's output messages.

1.2.2 The Administrator/Operator's View of a TP System

The administrator of this electronic mail system has users in Asia, Australia, the Americas, Europe, and Africa (see Figure 1.3). There are administrative staff on each of these continents. In addition, the system has gateways to other mail systems, and the administrator cooperates with the managers of those systems.

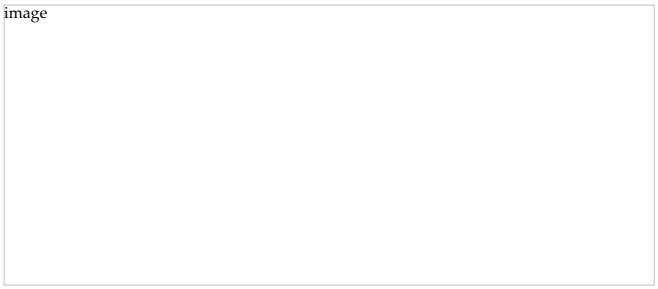


FIGURE 1.3 The system administrator/operator's view of the transaction processing system. This view consists of physical nodes containing hardware and interacting software modules. The application includes interfaces to facilitate administering and operating the system. The repository is a database that records the system configuration. The administrator views the system through the repository.

The primary functions of the administrator are to add and delete users, provide them with documentation and training, and manage the system security. The security and administration interface provides forms and transaction programs that allow the administrator to add users, change passwords, examine the security log to detect violations, and so on. These administrative transactions are ACID operations on the user database and the security database. They are just another aspect of the application and are programmed with the same high-level transaction processing tools used by other application designers.

Most of the software and hardware of a mail system are standard, but there are a few custom features that each organization has added. In particular, let us assume that the mail system software is a turn-key application provided by the vendor or by some third-party software house. The administrator has to plan for the installation of new software from the vendor as well as for installation of software from his own application development group. Such software upgrades and changes must be put through a careful integration, quality assurance, field testing, and then incremental installation on the various nodes of the network.

Besides these tasks, the system administrator monitors system performance and plans equipment upgrades as mail messages consume more storage and as increasing mail volume consumes disk space, processors, and network bandwidth. Once the system is installed, the administrator gets periodic reports on system utilization, indexed by component and by user. Again, these are standard applications that use the output from a performance monitor to populate a performance database; they use a report writer to generate formatted displays of the data.

The system administrator has a daunting job: He must manage thousands of users and myriad hardware and software components. He needs the help of a computer to do this. The administrative interface is an integral part of any transaction processing system and application. Like any other transaction processing application, it is programmed to provide a forms-oriented transactional interface to the system configuration. The system configuration is represented as a database, called the *repository* or *system dictionary*. The administrator can alter the configuration using standard transactions, inquire about the configuration using standard reports, or read the configuration using a non-procedural language, such as SQL, to generate specialized reports. The repository concept is developed in more detail in Subsection 1.3.2. The key point is that the system configuration should be represented as a database that can be manipulated using the high-level software tools of the transaction processing system.

While the system administrator makes policy decisions and plans system growth, the system operator performs the more tactical job of keeping the system running. He monitors system behavior, and when something breaks, he calls the repair man. Additionally, he performs operational tasks such as installing new software, connecting new hardware, producing periodic reports, moving archival media off site, and so on. There is an increasing tendency to automate operations tasks, both to reduce errors and to reduce staff costs.

Table 1.4 gives a sense of the scale of administration and operation tasks on systems with thousands or hundreds of thousands of components. If each terminal has a mean-time-to-failure of three years, then the operator of a large system will have to deal with a terminal failure about every 15 minutes. Similarly, he will have to deal with one disk failure a week and one processor failure per week. Just diagnosing and managing these events will keep an operations staff busy.

Table 1.4

Representative counts of the components of small and large transaction processing systems.There are so many components in a system that a special database (called the*repository*) is needed to keep track of them and their status. Notice especially the hundreds of thousands of terminals, tapes, and source files.

Hardware	Small/Simple	Medium	Large/Complex
Terminals or users	100	10,000	100,000
Hosts	1	10	100
Disks (capacity)	10 (= 10GB)	100 (= 100GB)	1K (=1TB)
Archive tapes (capacity)	1K (= 1TB)	10K (= 10TB)	100K (= .1PB)
Software			
Transaction programs, reports, screens, database files	400	4,000	40,000
Source and old versions of programs, reports, and screens	1,000	10,000	100,000
Domains/ fields (within tables)	1,000	10,000	100,000

Perhaps the most important message in Table 1.4 is that a transaction processing system has thousands—up to millions—of components. Tracking these components, diagnosing failures, and managing change is a very complex task. It is structured as a transaction processing application, with a database (the repository) and a collection of well-defined operations that transform the database. Other operations produce reports on the current system configuration and how it got to that state.

The system administrator, system operator, and application designer also face performance problems. The system often runs both *interactive transactions*, which are processed while the client waits for an answer, and *batch transactions*, which are submitted but may be processed later. Interactive transactions are necessarily small, since they must be processed within the few seconds that the client is willing to wait. If the system is overloaded or poorly configured, response times can increase beyond an acceptable level. In this case, the operator and the administrator are called upon to *tune* the system for better performance. As a last resort, the administrator can install more equipment. Increasingly, this tuning process is being automated. Most systems provide tools that measure performance, recommend or perform tuning actions, and recommend the type and quantity of equipment needed to process a projected load.

Estimating the performance of a TP system is difficult because the systems are so complex and because their performance on different problems can be very different. One system is good at batch jobs, another excels on interactive transactions, and yet a third is geared toward information retrieval applications (reading). Gradually, various usage patterns are being recognized, and standard benchmarks are being defined to represent the workload of each problem type.

An industry consortium, the Transaction Processing Performance Council (TPC), has defined three benchmarks. With each of these benchmarks, named A, B, and C, comes a scaleable database and workload, allowing the benchmarks to be run on computers, networks, and databases of every size, from the smallest to the largest. They can also be scaled up to the largest computers, networks, and databases. Using these benchmarks, each system gets three throughput ratings, called *transactions-per-second* (tps-A, tps-B, and tps-C). As the tps rating rises, the network size and database size scale accordingly.

These benchmarks also measure system price/performance ratio by accounting for the five-year price of hardware, software, and vendor maintenance. This five-year price is then divided by the tps rating to get a price/performance rating ($/tps).

The three benchmarks can be roughly described as follows:

TPC-A is a simple banking transaction with a 100-byte message input, four database updates, and a 200-byte message output to the terminal. It includes simple presentation services. The database operations are a mix of main-memory, random, and sequential accesses. The system definition and price includes ten terminals per tps.

TPC-B is TPC-A with the terminals, network, and two-thirds of the long-term storage removed. It is a database-only benchmark designed to give high tps ratings and low $/tps ratings to database systems. Its price/performance rating is typically ten times better than the more realistic TPC-A rating.

TPC-C is an order entry application with a mix of interactive and batch transactions. It includes realistic features like queued transactions, aborting transactions, and elaborate presentation services. As a group, the TPC-C transactions are about ten times more complex than TPC-A; that is, TPC-C tps ratings are ten times lower for the same hardware. TPC-C has just been approved as a standard. It will likely supplant TPC-A and TPC-B as an important performance metric.

TPC-C approximates a typical interactive transaction workload today. As computers and networks get faster, and as memories grow in size, the typical transaction is likely to grow larger and more complex than those in TPC-C. Table 1.5 gives a rough description of the cpu, I/O, and message costs of TPC-A, TPC-C, and a hypothetical large batch program. The TPC will likely define such a batch workload as TPC-D. The specifications for the TPC benchmarks and several other benchmarks are in the *Benchmark Handbook for Database and Transaction Processing Systems* [Gray 1991].

Table 1.5

Typical performance measures of interactive transactions. Interactive transactions respond within seconds. By contrast, batch transactions can run for hours or days and thus consume huge amounts of data and processing.

Performance/transaction	Interactive transactions only		
	Small/Simple	Medium	Large/Complex
Instructions/transaction	100K	1M	100M
Disk I/O/transaction	1	10	1000
Local Messages (bytes)	10 (5KB)	100 (50KB)	1,000 (1MB)
Remote Messages (bytes)	2 (300B)	2 (4KB)	100 (1MB)
Cost/transaction/second	10k$/tps	100k$/tps	1M$/tps
Peak tps/site	1,000	100	1

1.2.3 Application Designer's View of a TP System

Now let us look at the sample mail system from the perspective of the application designer or application implementor. In the past, the roles of application designer and application implementor were separate. With the advent of application generators, rapid prototyping, and fourth-generation programming languages, many implementation tasks have been automated so that the design *is* the implementation.

Applications are now routinely structured as *client* processes near the user, making requests to *server* processes running on other computers. In the mail system, the client is the mail program running on Andreas' workstation, and the server is running on the Berlin host (see Figure 1.3). Of course, the client could be a gas pump, a warehouse robot, or a bar code reader. In all cases, the concepts would be the same.

The client program executes as a process in the user's workstation, providing a graphical and responsive interface to the human user. This aspect of the application, called *presentation services*, gathers data and navigates among the forms. If access to other services is required, the client passes the request to a server running on a local or remote host, as shown in Figure 1.6.

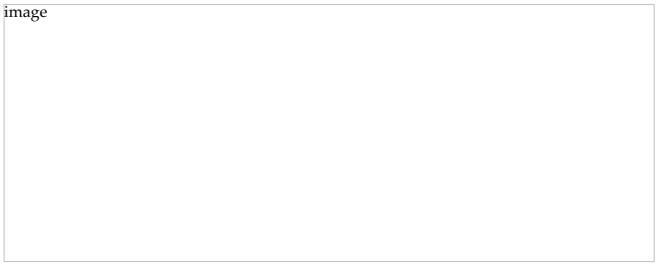
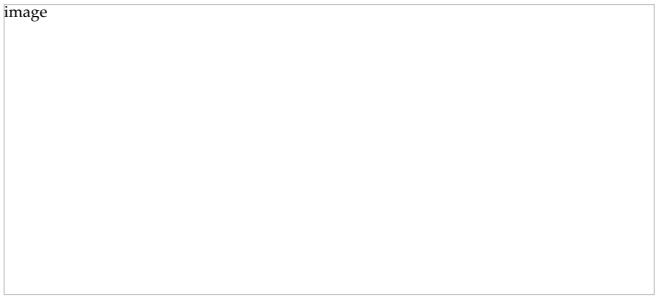


FIGURE 1.6 **The structure of a forms-oriented application in which presentation services are performed in the client.** The client program was generated by an application generator. The client actions generate remote procedure calls to the TP monitor running in the shared host computer that stores the mailboxes and messages. The TP monitor launches one or another services to perform the requested operations against the database. Of course, the host could be geographically distributed (as in Figure 1.3), and the client could store part of the database and perform some service functions locally.

The request to the server (service) arrives at the host as a message directed to the transaction processing monitor at that host. The TP monitor has a list of services registered with it. In the mail system example, these services are application-specific functions, such as logon to the mail system, read a message, delete a message, send a message, and so on. When a service request arrives, the TP monitor looks up the service in its repository and checks that the client is authorized to use that service. The TP monitor then creates a process to execute that service, passing the text of the initial request to the server process. The server process then performs its function, accessing the database and conversing with the client process. If the service is very simple, it accesses its database, replies, and terminates. If the service is more complex, it replies and waits for further requests from the client. One service may call on other services and may implicitly access remote services by accessing data stored at remote sites.

What did the application designer do to produce the client and server code? The client code was probably built with an application generator that produced a subroutine for each input/output form shown in Figure 1.2. The designer painted each form, defining input fields, output fields, and buttons that navigate to other forms. For example, the logon form gathers the client name and password, then passes them to the host, invoking the mail server logon service via a remote procedure call. The service tests the password and returns the result of the test to the client. If the logon is successful, the service will have allocated a *context* for the client on the server, so that subsequent calls can quickly refer to the client's mailbox and status on the host. The logon form then invokes the header form, which in turn calls the header service on the mail server host to enumerate the recent incoming and outgoing messages. The generated code for the logon client has a format something like this:



This client code is represented as the Logon box in the workstation of Figure 1.2. It is written in a programming language with standard syntax and control flow, but it also has the screen handling functions of a presentation service (the ACCEPT and DISPLAY verbs), remote procedure call functions that allow it to invoke services at any node in the network (the CALL verb), and transaction verbs that allow it to declare transaction boundaries (Begin and Commit verbs).

The client programming language is often *persistent*. This means that if the client process fails or if the transaction aborts, the process and its persistent state will be reinstantiated as of the start of the transaction. For example, if the power fails and the processor restarts, then the client process will be recreated as if the transaction had simply aborted. Persistent programming languages tie the application state together with the database state and the network state, making changes to all aspects of state atomic—all-or-nothing.

The server code may also have been produced by an application generator. For example, the logon server has to check that the user password is correct and then *open* the user's mailbox, thereby producing a *token* that will identify this client mailbox on subsequent calls. This application logic is so simple that there may be a template for it already present in the system. If not, the application designer will have to write a few lines of SQL and C code to perform these functions.

To summarize, the transaction processing system presents the application designer with a distributed computation environment and with a collection of application-generation tools to speed and simplify the design and implementation task. These application generators are used both for rapid prototyping and developing the actual application. The application designer either paints screens or writes code fragments that are attached to clients or services. All these screens, clients, and services are registered in the system repository. When an application executes, the client invokes services via a transactional remote procedure call mechanism.

1.2.3.1 Transactional Remote Procedure Calls

What is a transactional remote procedure call mechanism? It is best to develop this idea in stages. A *procedure call mechanism* is a way for one program to call another—it is the subroutine concept, a familiar part of virtually all programming languages. In general, all procedures are part of one address space and process. *Remote procedure calls* (RPCs) are a way for one process to invoke a program in another remote process as though the subroutine in the remote process were local. It appears to both the caller and the callee that they are both in the same process; the remote procedure call looks like a local procedure call.

Remote procedure calls, however, need not be remote. The client and the server can be in the same computer and still use the same interface to communicate. Local and remote procedure calls have the same interface, so that the client program can move anywhere in the network and still operate correctly. The client and the server need not know each other's location. Locating the client at a particular site is a performance and authorization decision. If the client interacts heavily with the input/output device, then the client belongs near the data source; if the client interacts more heavily with a single server, then the client probably belongs near the server.

Given these approximate definitions of procedure call and remote procedure call, we can now discuss the concept of *transactional remote procedure call*. Transactional remote procedure calls are a way to combine the work of several clients and servers into a single ACID execution unit. The application declares the transaction boundaries by calling Begin_Work(). This starts a new transaction and creates a unique transaction identifier (trid) for it. Once a transaction is started, all operations by the client are tagged by that transaction's identifier. The transaction identifier is sent with all service requests, so that the operation of that service is also within the scope of the transaction. When the transaction commits, all the participating services also commit the operations of that transaction. The commit logic is implemented by the TP monitors of each participating node. They all go through a wedding ceremony protocol (two-phase commit protocol) coordinated by a trusted process. Recall that the two-phase commit protocol is: Q: "Do you commit?" A: "I do." Q: "You are committed." A: "Great!" If anything goes wrong, everything rolls back to the start of the transaction.

In transactional systems, servers are generally called *resource managers*. A resource manager is the data, code, and processes that provide access to some shared data. Transaction processing systems make it easy to build and invoke resource managers. The mail system of our running example is a resource manager. RPC is the standard way for an application program to invoke a resource manager.

Transactional remote procedure call (TRPC) is just like remote procedure call, except that it propagates the client's transaction identifier, along with the other parameters, to the server. The TRPC mechanism at the server registers the server as part of the client's transaction. Figure 1.7 shows the flow of messages among clients and servers in a typical transaction. Each of the local and remote procedure calls in Figure 1.7 are tagged with the client's transaction identifier.

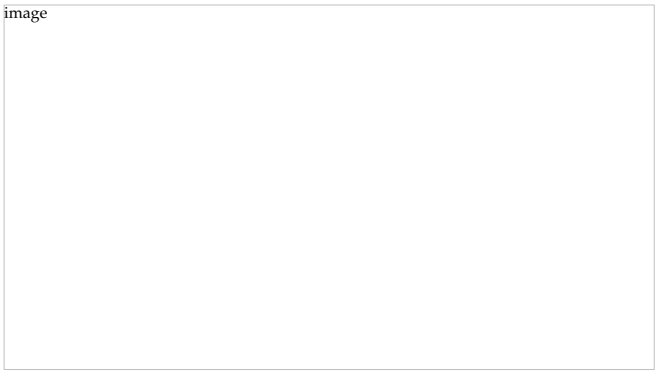


FIGURE 1.7 **The data flow among the active components of the distributed computation shown in**Figure 1.6. The user interacts with the screen interface. The client reads and writes the screen and communicates with servers via remote procedure calls (shown as bold lines). The initial invocation of the service is performed by the TP monitor that creates the server process. The server then interacts with the database and the client. The server may also make further calls to other servers and transparently access remote data via the database system. This figure shows a client calling a single server, but a client may invoke several services within one transaction. Client-server interactions can be structured as a single transaction or as a sequence of transactions.

1.2.3.2 Failure

What happens if something goes wrong in the middle of this? For example, what if the user hits the cancel key, or if the server decides that the input is wrong, or if there is a power failure, or if the server fails in some way, or if something else fails? If the failure happens *after* the entire operation completes, then the client and the server should remember the state changes, and both should return to the state at the end of the transaction—that is the durability property. On the other hand, if the failure happens *before* the operation completes, then some resource manager may not be able to commit the transaction. To be safe, the client and the server should return to their states as of the start of the transaction. All subsequent updates should be undone, the windows on the workstation should return to their original states, and the database should return to its original state—this the atomicity property.

The TP monitor automatically performs this reset, or reconstruction logic, in case of failure. The application program (client) declares the start of a computation that is to be treated as an ACID unit by issuing a Begin_Work() verb. When the computation is complete (when it is a correct transformation), the client issues a Commit_Work() verb. All actions of the transaction between the Begin_Work() and Commit_Work() verbs will be treated as a single ACID group. If the computation fails before the Commit_Work(), or if any part of the computation invokes the Rollback_Work() verb, then the state of the system will be returned to the situation as of the Begin_Work(). In particular, the client, the servers, the screen, and the database will all be reset to that state. This simplifies the exception handling for the application designer. All this undo work is automatic.

As explained later, many transaction systems allow *nested* transactions and partial rollback of a transaction. The presentation here begins with the simple model of*flat* transactions with no intermediate *savepoints*. Chapters 4 and 5 explore generalizations of this simple model.

1.2.3.3 Summary

A transaction processing system is an application development environment that includes presentation services, databases, and persistent programming languages. A TP system also provides transactional remote procedure calls, which allow the application to be distributed as clients and servers in a network of computers.

Transaction processing systems provide the application programmer with the following computational model:

Resource managers. The system comes with an array of*transactional resource managers* that provide ACID operations on the objects they implement. Database systems, persistent programming languages, queue managers (spoolers), and window systems are typical resource managers.

Durable state. The application designer represents the application state as durable data stored by the resource managers.

TRPC. Transactional remote procedure calls allow the application to invoke local and remote resource managers as though they were local. They also allow the application designer to decompose the application into client and server processes on different computers interconnected via transactional remote procedure calls.

Transaction programs. The application designer represents application inquiries and state transformations as programs written in conventional or specialized programming languages. Each such program, called a *transaction program*, has the structure:

In other words, the programmer brackets the successful execution of the program with a Begin-Commit pair and brackets a failed execution with a Begin-Rollback pair.

Consistency. The work within a Begin-Commit pair must be a correct transformation. This is the C (consistency) of ACID.

Isolation. While the transaction is executing, the resource managers ensure that all objects the transaction reads are isolated from the updates of concurrent transactions. Conversely, all objects the transaction writes are isolated from concurrent reads and writes by other transactions. This is the I of ACID.

Durability. Once the commit has been successfully executed, all the state transformations of that transaction are made durable and public. This is the D (durability) of ACID.

Atomicity. At any point before the commit, the application or the system may abort the transaction, invoking rollback. If the transaction is aborted, all of its changes to durable objects will be undone (reversed), and it will be as though the transaction never ran. This is the A (atomicity, all-or-nothing) of ACID.

The example mail system and its administrative and operations interface were programmed in this way as an application using the tools of the transaction processing system. Sample tools included the database system, the presentation management system, and the application generator. The next subsection gives a hint of how each of these tools is built and how they fit into the transaction processing system.

1.2.4 The Resource Manager's View of a TP System

We have seen thus far that a transaction processing system includes a collection of subsystems, called*resource managers*, that together provide ACID transactions. As application generators, user interfaces, programming languages, and database systems proliferate and evolve, resource managers are continually being added to the TP system.

The transaction processing system must be extensible in the sense that it must be easy to add such resource managers. There are many database systems (e.g., DB2, Rdb, Oracle, Ingres, Informix, Sybase), many programming languages (e.g., COBOL, FORTRAN, PL/I, Ada, C, C++, persistent C), many networks (e.g., SNA, OSI, TCP/IP, DECnet, LAN Manager), many presentation managers (e.g., X-windows, News, PM, Windows), and many application generators (CSP, Cadre, Telos, Pathmaker). Each customer seems to select a random subset from this menu and build an application from that base. A transaction processing system facilitates this random selection. The TP system provides a way to interconnect applications and resource managers, while providing the ACID properties for the whole computation.

The key to this resource manager interoperability is a standard way to invoke application services and resource managers. Such a mechanism must allow invocation of both local and remote services. As explained in the previous section, the standard mechanism for doing this is a remote procedure call. The transactional remote procedure call is an extension of the simple procedure call. It allows the work of many calls and servers to be coordinated as an ACID unit.

The resulting computation structure looks like a graph, or tree, of processes that communicate with one another via these transactional remote procedure calls. The TP system provides the binding mechanisms needed to link the client to servers. The linkage could be local or remote:

Local. Client and server are in the same address space or process, in which case the server code is bound to the client code and the invocation is a simple procedure call.

Remote. Client and server are in different processes, in which case the invocation consists of the client sending a message to the server and the server sending a reply to the client. This is a remote procedure call.

The caller cannot distinguish a local procedure call from a remote procedure call; they have the same syntax. Local calls generally have better performance, but remote calls allow distributed computation and often provide better protection of the server from the client. Figure 1.8 illustrates the typical call structure of an application invoking various application services and various resource managers. Each box is optionally a process, and the connection among boxes is via a local or remote procedure call.

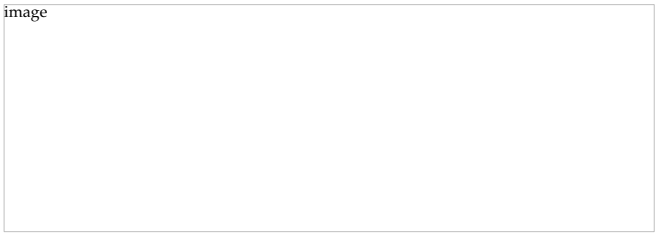


FIGURE 1.8 **The execution of a transaction is spread among application programs (clients and servers) and resource managers**Services and resource managers can be invoked by local or remote procedure calls. Local services and servers are bound to the caller's address space, while remote ones run in separate processes, perhaps on remote computer systems. The entire computation occurs within the scope of a single transaction. The transaction manager monitors the progress of transactions, connects clients to servers, and coordinates the commit and rollback of transactions.

Aside from managing the creation and intercommunication of processes performing the transaction, each TP monitor has a set of core services that it provides to the resource managers. These services help the resource managers implement ACID operations and provide overall execution control of the application program that invokes the individual resource managers.

The basic control flow of an application is diagrammed in Figure 1.9. The Begin_Work() verb starts the transaction, registering it with the*transaction manager* and creating a unique transaction identifier. Once the application has started a transaction, it can begin to invoke resource managers, reading and writing the terminal as well as databases, and sending requests to local and remote services.

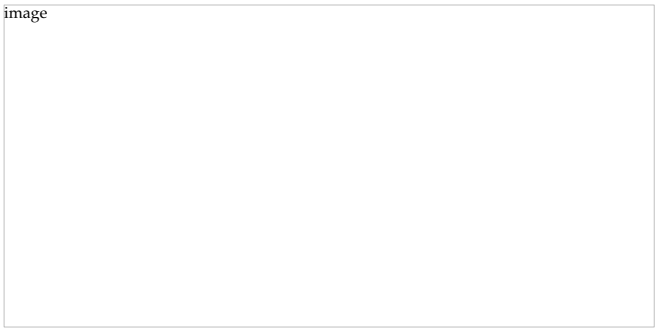


FIGURE 1.9 **The role of the core services (transaction manager, log manager, and lock manager) in the execution of a transaction.** The transaction scheduler (not shown) sets up the processes that execute the transaction.

When a resource manager gets the first request associated with that transaction, it *joins* the transaction, telling the local transaction manager that it wants to participate in the transaction's commitment and rollback operations. It is typical for several resource managers to join the transaction. As these resource managers perform work on behalf of the transaction, they keep lists of the changes they have made to objects. As a rule, they record both the old and the new value of the object. The transaction processing system provides a logging service to record these changes. The *log manager* efficiently implements a sequential file of all the updates of transactions to objects. Of course, the other resource managers have to tell the log manager what these updates are.

To provide isolation, resource managers *lock* the objects accessed by the transaction; this prevents other transactions from seeing the uncommitted updates of this transaction and prevents them from altering the data read or written by this uncommitted transaction. The transaction processing system provides a *lock manager* that other resource managers can use.

When the transaction issues Commit_Work(), the transaction manager performs the two-phase commit protocol. First, it queries all resource managers that joined the transaction, asking if they think the transaction is a consistent and complete transformation. Any resource manager can vote no, in which case the commit fails. But if all the resource managers vote yes, then the transaction is a correct transformation, and the transaction manager records this fact in the log, informing each resource manager that the transaction is complete. At this point, the resource managers can release the locks and perform any other operations needed to complete the transaction.

If the transaction should fail during execution, or if a resource manager votes no during phase 1 of the two-phase commit, then the transaction manager orchestrates transaction rollback. In this case, the transaction manager reads the transaction's log and, for each log record, invokes the resource manager that wrote the record, asking the resource manager to undo the operation. Once the undo scan is complete, the transaction manager invokes each resource manager that joined the transaction and tells it that the transaction was aborted.

The transaction manager also orchestrates transaction recovery if a node or site fails. It provides generic services for the failure of a single object, the failure of a resource manager, and the failure of an entire site. The following paragraphs sketch how the transaction manager helps in system recovery.

If a site fails, the TP system restarts all the resource managers. Several transactions may have been in progress at the time of the failure. The resource managers contact the transaction manager as part of their restart logic. At that time, the transaction manager informs them of the outcome of each transaction that was active at the time of the failure. Some may have committed, some may have aborted, and some may still be in the process of committing. The resource manager can recover its committed state independently, or it can participate in the transaction manager's undo and redo scan of the log.

If a resource manager fails but the rest of the TP system continues operating, the transaction manager aborts all uncommitted transactions involved with that resource manager. When the resource manager returns to service, the transaction manager informs the resource manager about the outcome of those transactions. The resource manager can use this information and the transaction log to reconstruct its state.

If a particular object is lost but the resource manager is otherwise operational, then the resource manager can continue to offer service on other objects while the failed object is reconstructed from an archive copy and from a log of all committed changes to that copy. The transaction manager and the log manager aid recovery from an archive copy of the object.

Each site usually has a separate transaction manager. This allows each site to operate independently of the others, providing *local autonomy*. When the transaction's execution is distributed among several sites, it is distributed among several transaction managers. In that case, the two-phase commit protocol for multiple processes generalizes easily to multiple transaction managers.

As explained in Subsection 2.7.4, the X/Open consortium draws Figure 1.9 differently. It assumes that each resource manager has a private log and a private lock manager, and X/Open assumes that the resource manager performs its own rollback, based on a single rollback call from the transaction manager, in case the transaction aborts. The resulting simpler picture is shown in Figure 1.10.

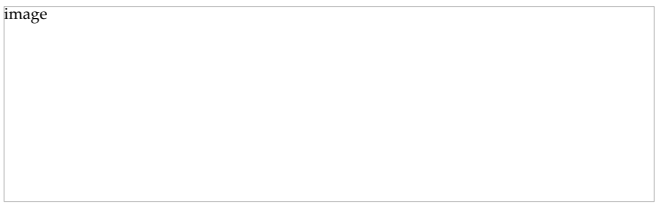


FIGURE 1.10 **The X/Open transaction processing model.** As in Figure 1.9, the application starts a transaction managed by the local transaction manager. As the application invokes resource managers, they join the transaction. When the application commits or aborts, the transaction manager broadcasts the outcome to the resource managers. In this model, the resource managers have private locks and logs, and the transaction manager does not provide an undo scan of the transaction log.

Figures 1.9 and 1.10 describe centralized transactions involving a single transaction manager. When multiple transaction managers are involved, the figure becomes more complicated since the transaction managers must cooperate in committing the transaction. Figure 1.11 diagrams this design. More detailed discussions of the X/Open model are found in Subsections 2.7.4 and 16.6.

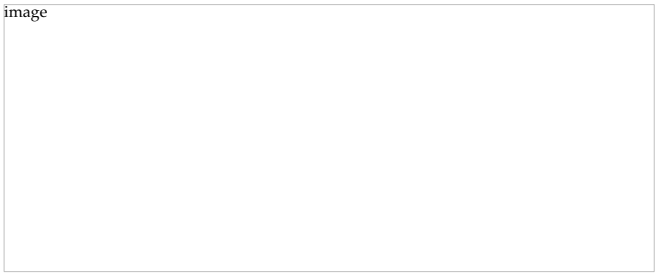


FIGURE 1.11 **The X/Open distributed transaction processing model.** As in Figures 1.9 and 1.10, the application starts a transaction managed by the local transaction manager. When the application, or some resource manager acting on behalf of the application, makes a remote request, the communications managers at each node inform their local transaction managers of the incoming or outgoing transaction. Transaction managers at each node manage the transaction's work at that node. When the transaction commits or aborts, the transaction managers cooperate to provide the atomic and durable commit.

1.2.5 TP System Core Services

To summarize, the following are the *core services* of the transaction processing monitor:

Transactional RPC. Authorizes, schedules, and invokes the execution of services (servers).

Transaction manager. Orchestrates the commit and rollback of transactions as well as the recovery of objects, resource managers, or sites after they fail.

Log manager. Records a log of changes made by transactions, so that a consistent version of all objects can be reconstructed in case of failure.

Lock manager. Provides a generic mechanism to regulate concurrent access to objects. This helps resource managers provide transaction isolation.

Resource managers extend the transaction processing system by using these core services. Generally, there is a separate TP system at each site or cluster of a computer network. These TP monitors cooperate to provide a distributed execution environment to the user.

The bulk of this book is dedicated to explaining the concepts of each of these core services and how they are implemented. It also explores how database system resource managers use these facilities. Before diving deeper into the TP system implementor's view, let us return to the top-level view of the transaction processing system.

1.3 A Transaction Processing System Feature List

Rather than viewing the system from the perspective of a particular user, we can look at the typical features of transaction processing systems. The most obvious of these features are the application design and generation tools that, in turn, are built around the repository. Underlying the application generators and the repository are a database system used to store persistent data (often an SQL database) and a data communications system used to provide an abstract interface to terminals, workstations, and other input-output devices and remote requestors. Behind this DataBase-DataCommunications (DB-DC) layer is the TP monitor itself, with its core services. Peripheral to these mainline features are ancillary tools used to administer, tune, and operate the system. Of course, this all runs on a substrate consisting of an operating system, a network, and computer hardware. The feature list here focuses on application generation, the repository, the database, data communications, transaction management, and operations.

1.3.1 Application Development Features

The goal and promise of application generators is to automate programming by translating designs directly into executable systems. Their major success has been in the transaction processing area. Typical projects report that 90% of clients and servers are automatically generated via point-and-click graphical programming interfaces. Application generators allow rapid prototyping, reduce project risk, and improve system usability. What this really means is that application generators are able to generate most of the generic functions of the application from a standard toolbox, thereby leaving complex domain-specific issues (such as discounts, tax codes, and work scheduling) to conventional programming. This residual or core code is usually written in a standard programming language (e.g., COBOL, C, FORTRAN, SQL).

In addition, there are tools that analyze a schematic description of the database design and ultimately produce an optimized set of SQL data-definition statements to represent the database. Screen design aids allow the designer to *paint* the screens associated with each transaction. The resulting graphics are translated to programs that read and write such forms.

Once the application has been generated in this way, tools are provided to populate the database with synthetic data and to generate a synthetic load on the system. Tracing and debugging tools are used to test and certify the correctness of the programs.

Most such systems provide a *starter system*, a simple application that nevertheless includes fairly comprehensive administrative and operations functions, a security system, an accounting system, and operations procedures. These preprogrammed tools are then applicable to new applications built with the application generator. They also address the important issues of operator and administrator training.

1.3.2 Repository Features

The repository holds descriptions of the objects in the system and records the interdependencies among these objects. Repositories are also called *dictionaries* or *catalogs*. The catalogs of SQL database systems provide a concrete example of a repository: They track the database tables, indices, views, and programs. The SQL system catalog also tracks the dependencies among these objects; for example, this program uses that view, which in turn uses those indices and tables. When someone changes a table, the SQL system automatically recompiles all the affected programs. Before making the change, the administrator can run a standard report to display the programs that will be affected by the change. All this is inherent in the SQL catalogs. The shortcoming of this example is that the SQL system is only aware of the database part of the application. If the change should propagate to a client, the SQL system will not know about it.

The need for a repository is implicit in Figure 1.3 and Table 1.4. The typical system has thousands of terminals and users and hundreds of applications, screens, tables, reports, procedures, and archive copies of these objects. When old versions of applications, procedures, and other objects are considered, the number of objects grows well beyond 10,000. Remembering this information is difficult, and changing it is error-prone, because any change is likely to involve changing many system components. For example, adding a field to a screen alters the screen, probably alters the client software and the server interface, and probably adds a column to a table. All these changes must be made together (as an ACID unit). The change should be expressed once and should automatically be propagated to each consumer of the object. If the changes

do not work out, then they should *all* be reversed, so that the system returns to its previous state. Implementing such a change manually is a recipe for disaster. Chapter 3 points out that such operational mistakes cause many system failures.

The operations problem is a classic inventory and bill-of-materials problem.[2] Changing a system is analogous to building a new version of a complex mechanism. The manufacturing industry learned long ago that computers can automate most of the bill-of-materials logic. The computer industry is just learning that the same techniques apply to manufacturing software systems.

Take the UNIX MAKE facility, in which a root file points to a list of subsidiary files is a bill-of-materials for the root. To make a root, one must make all the subsidiary files. These files, in turn, are MAKE files for subcomponents of that root. This is an example of bill-of-materials done in an unstructured way. The bill-of-materials is represented as ASCII text that is a free-form representation of the information. Because the information is not structured in a uniform way, it is hard to find all the components that use a particular subsystem, it is hard to find all versions of a particular subsystem, and there is no connection between the bill-of-materials and the end-product. That is, once you MAKE something, the result is not attached to its bill-of-materials; put another way, there is no version management.

Here, then, is the key idea of the repository (seeFigure 1.12):

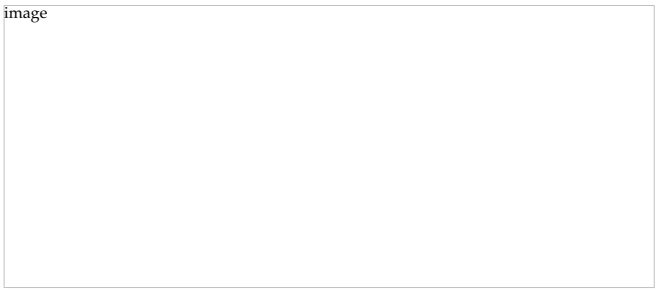


FIGURE 1.12 **The repository describes the system state in a durable, secure, active database.**This database is changed by transactions that alter the repository, along with the system state. The desirable properties of a repository are listed at the right.

Design. Capture all application design decisions in a database.
Dependencies. Capture all application design and module dependencies in a database.
Change. Express change procedures as transactions on this database.

This is not a new idea; it is just an application of the bill-of-materials idea to a new domain. Capturing all design decisions in structured form (i.e., machine-readable form) reduces errors and simplifies maintenance. Automating change reduces errors and improves productivity.

A good repository needs to be*complete*. It needs to describe all aspects of the system—not just the database, the programs, or the network. The repository stores the system bill-of-materials. To be complete, the repository must be *extensible*: It must be possible to add new objects, relationships, and procedures to it. The application designer must be able to define a new kind of object without re-implementing the repository. The true test of extensibility is whether the repository has a core that implements the repository objects, with extensions that implement the other objects (such as screens, tables, clients, and servers). For example, the SQL catalogs should just be an extension of the repository. If the vendor has extended the repository in this way, using the repository's built-in extension mechanism, then the application designer will probably be able to make comparable extensions using the same mechanism.

A repository should be *active*; that is the description of the object should be consistent with the actual state of the object. For example, if a domain is added to a file, or if a field is added to a screen, then the description of the object should reflect this change. Repositories without this property are called *passive*.

Active repositories are essential, but also very difficult to implement efficiently. The active repository concept, taken to its limit, requires that each time a record is added to a sequential file, the file's record-count be updated in the catalog. Doing this would add a random update to every sequential insert—an overhead of several hundred percent. Even more troublesome, the repository is often built on top of the operating system, file system, and network system. With this *layered* and *portable* design, some changes to the operating system configuration, file system, or network system may not be registered with the repository layered above them. For example, replacing the existing COBOL compiler with a new version of the compiler could be disastrous for the application: the new version may have a new bug. If such a change is done via the repository, the old compiler will be saved for the application. If the change is done directly via the file system, without involving the repository, the change will cause the application to fail. To be truly active, the repository must register the dependency of the application on the old COBOL compiler and prevent its deletion from the system until the associated applications are also deleted.

If the application is distributed among autonomous sites, then the repository must also be distributed to provide*local autonomy*: the ability to operate on local objects, even though the site is disconnected from the others. If the repository is truly active, then a site cannot operate without it. For example, files cannot be created, altered, or deleted if the repository is unavailable (down). Thus, the repository must always be available. To achieve this, the descriptors and dependencies of all objects stored at a site must also be stored at that site. If an object is replicated or partitioned among many sites, then its descriptors must be replicated at each site. If an object at one site depends on an object at another site, then the dependency must be registered at both sites.

The repository is one database. It is distributed, partitioned, and replicated, and it has some interesting transactions associated with it. It is a relatively small database (less than a million records), but it has high traffic. Bottlenecks and other performance problems have plagued many repository implementations to date. Thus, a repository must be *fast*, so that it does not hinder performance.

Using a database representation for the repository gives programmatic access to the data for ad hoc reports and provides a report writer to produce standard reports on the system structure and status. In addition, the database provides durable storage with ACID state transformations (transactions). Thus, these reporting and durability requirements are automatically met. Since the repository holds all the information about the system, it must be *secure*, so that unauthorized users and programs cannot read or alter it.

To summarize, the repository is a database that describes the *whole* system, along with programs to manage that database. The repository should be*complete*, *extensible*, *active*, *secure*, and *fast*. It should provide standard reports and allow programs to access the data in ad hoc ways subject to *security* restrictions. If the repository is distributed, then it should provide*local autonomy*. Such a repository provides the basis for application development tools, system administration tools, and operations tools (see Figure 1.12).

No one has yet built a proper repository. Application generators come closest. Their repositories often meet the requirements of being active on the objects they manage, and they are extensible, accessible, secure, and fast. But they are not complete. They capture only design information of the application, not the operational information of the system, such as files, disks, networks, and information hiding in system configuration files. Since the latter objects represent reality, they can change without the design repository sensing the change. In that respect, design repositories are not active. In fact, an unhealthy split has evolved between design repositories and operational repositories.

Two of the most advanced repositories are CDD/R from Digital and Pathmaker from Tandem. CDD/R is the basis for Digital's transaction processing system (ACMS), its software development environment (Cohesion), and its database system (Rdb). Pathmaker is an application generator for Tandem's Pathway system. Figure 1.13 shows most of the database schema for the Pathmaker repository. (Not shown is the authorization database or the SQL repository; these would add another 20 tables to the 34 shown in the diagram.) Each box represents a database table. The tables describe people, projects, the system configuration, screens, clients (requesters), services, and how they are packaged into servers. The tables also describe the message formats for client-server interaction.

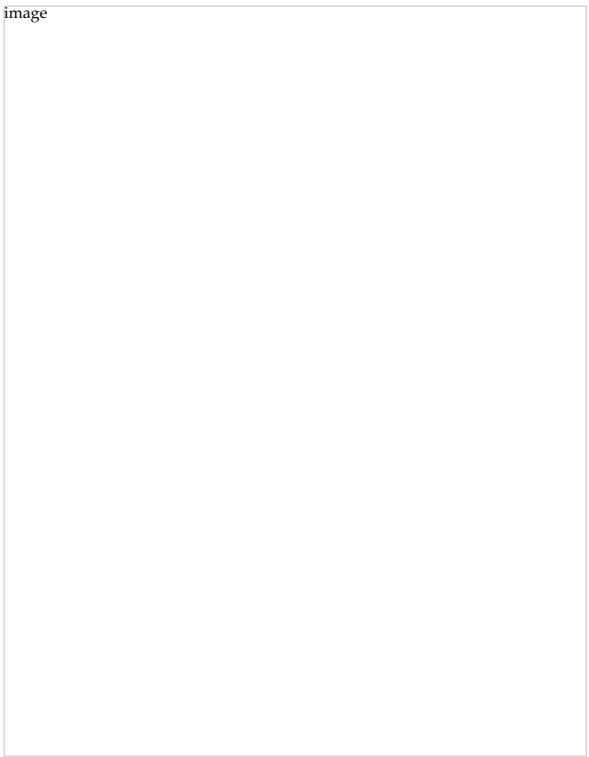


FIGURE 1.13 The tables of Tandem's Pathmaker repository.

1.3.3 TP Monitor Features

The TP monitor provides an execution environment for resource managers and applications. When requests arrive from local or remote clients, the TP monitor launches a server to perform the request. If the request arrives with a transaction identifier, then the server becomes part of that pre-existing transaction. If the request is not already transaction protected, then the TP monitor or server starts a new transaction to cover the execution of the request. Thus, the TP monitor provides a transactional RPC mechanism.

Before creating a server for a request, the TP monitor authorizes the client to the service. The client has usually been authenticated by the TP monitor as a particular person or group of persons (e.g., all stock clerks). The TP monitor checks that the client is authorized to invoke that service. This authorization check is stated as a function of the client group, client location, and time. For example, local stock clerks may run the invoice transaction during business hours. Optionally, the TP monitor records the security check or security violation in an audit trail.

If the client is authorized to invoke the service, then starting the service becomes a scheduling decision. If the system is congested, the start may be delayed. The TP monitor may have a preallocated pool of server processes and may assign them to requests on demand. Such a process pool is called a *server class*. If all members of the server class are busy, the request must wait or a new server process must be allocated. The TP monitor may create a server, put the request in a server class queue, or even put the request in a queue for later (e.g., overnight) service. This server class scheduling and load balancing is a key responsibility of the transaction processing monitor.

Once the process is scheduled and is executing the request, the service can invoke other resource managers and other services that, in turn, join the transaction. The transaction manager tracks all servers and resource managers joined to the transaction and invokes them at transaction commit or rollback. If the transaction is distributed among many sites of a computer network, the transaction manager mediates the transaction commitment with transaction managers operating at other sites.

The application view of transactions is that a transaction consists of a sequence of actions bracketed by a Begin-Commit or Begin-Rollback pair. The Begin allocates a transaction identifier (trid). Each subsequent

action is an operation performed by some resource manager as either a local or a remote procedure call. These procedure calls implicitly carry the trid. All operations on recoverable data generate information (log data) that allows the transaction to be rolled back (undone) or atomically and durably committed. In addition, to isolate the transaction from concurrent updates by others, each resource manager ordinarily acquires locks on the objects accessed by the transaction. The trid is the tag used for locks and log records. It is the ACID object identifier.

Any process executing on behalf of the transaction will have the transaction's trid. If the process is running on a transaction-oriented operating system, the trid is part of the process state. If transactions have been grafted onto the operating system by the TP monitor's RPC mechanism, then the trid is a global variable of the process.

Transactions can declare *savepoints*—points within the transaction execution to which the application can later roll back. This partial rollback creates the ability for servers or subroutines to raise and process exceptions cleanly. A savepoint is established when the invocation starts. If anything goes wrong, the service can roll back to that savepoint and then return a diagnostic. The server can tell the client that the call failed and was a null operation. This provides simple server error semantics without aborting the entire transaction. Applying this concept more broadly, the work after a savepoint looks like one transaction *nested* within another. The subtransaction can abort independently of the parent transaction, but the transaction can commit only if all its ancestor transactions commit.

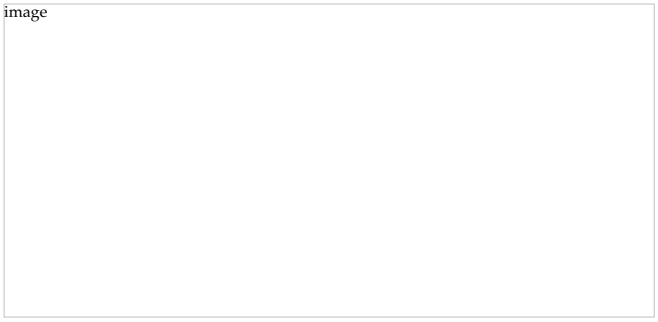


FIGURE 1.14 **The different types of transaction executions**. In the normal case (≈97%) the transaction executes and commits. In some cases (≈3%) the transaction aborts, either it calls rollback or it is rolled back by the system. In other cases, the transaction does a partial rollback to an intermediate savepoint and then continues forward processing.

In summary, a TP monitor authorizes client requests to servers, invokes servers, and provides a transactional execution environment to servers and resource managers so that they can implement ACID operations on durable objects. Besides managing individual transactions, the transaction manager also manages the resource managers. It invokes the resource managers at system startup and system checkpoint, and it coordinates the recovery of resource managers at system restart. It also orchestrates the recovery of objects from archival storage.

1.3.4 Data Communications Features

In discussing data communications, one has to consider the*classic world* of mainframes driving dumb terminals—terminals with no application logic—and the*client-server world* of clients doing all presentation management and driving servers via transactional RPCs (TRPCs). Technology is quickly making the classic approach obsolete; workstations, gas pumps, and robots are now powerful, self-contained computers. The older transaction processing systems, rooted in the classic design, are evolving to handle the new client-server approach depicted in Figure 1.6. Meanwhile, a new generation of transaction processing systems designed for the client-server model is emerging.

In both the classic and client-server designs, data communications provides an application programming interface to remote devices. The data communications software runs atop the native operating system and networking software of the host computer. There are many kinds of computer networks and networking protocols. The TP system's data communications subsystem provides a convenient and uniform application programming interface to all these different network protocols. In the OSI model, data communications is at the application or presentation layer (layer 6 or 7).

The data communications system is charged with reliable and secure communications. To prevent forgery or unauthorized disclosure, security is generally achieved via message encryption. Reliability is achieved by recording each output message in durable storage and resending it until it is acknowledged by the client. To avoid message duplication, each message is given a sequence number. The client acknowledges, but ignores, duplicate messages.

The ideal data communications system provides a uniform interface to the various communications protocols, so that higher layers do not need to be aware of the differences among the standards[3]. Applications want a single form of remote procedure call and do not want to learn about all the different ways networks name their transactions (SNA/LU6.2, OSI/TP, IMS, Tandem TMF, DECdtm, and Tuxedo all have different naming schemes.) The gateway process interfacing to that world masks these differences and translates them into the local standard interface. The data communications software provides this layer and, in doing so, provides a uniform interface to the plethora of network protocols. The resulting picture looks something like Figure 1.15.

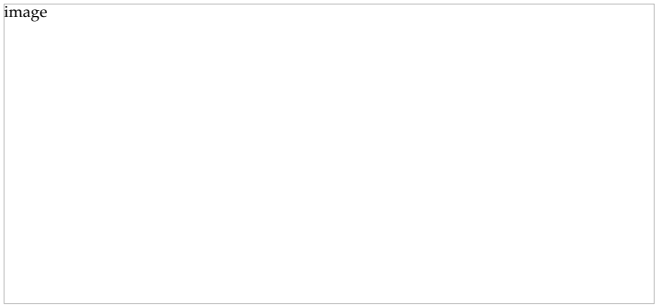


FIGURE 1.15 **The data communications subsystem and presentation services provide a uniform interface to the data communications protocols used by clients and terminals.** This protocol must add many features to simple protocols, like SNA/LU0 or TCP/IP, but needs to add less functionality to a protocol stack like SNA/LU6.2 or OSI/TP.

1.3.4.1 Classic Data Communications

In a classic data communications system, the terminal is a forms-oriented device, a gas pump, or a bar-code reader with little or no internal logic. All the presentation services logic (formatting of the display) is managed by the application running in the host. There is extreme diversity among terminal types and sub-types. The IBM 3270 terminal, for example, came with small and large screens, color, many different fonts and keyboards, a lightpen, and a printer. These options made it very hard to write generic applications; thus, *device independence* became the goal of the data communications subsystem. Over the last 30 years, transaction processing systems have evolved from cards to teletypes to character-mode displays to point-and-click multimedia user interfaces (see Table 1.16). Application designers want to preserve their programming investment; they want old programs to run on new input/output devices. Thus, teletypes emulated card readers and card punches (the source of 80-character widths). Today, many window systems emulate teletypes emulating card readers. Is this progress? No, but it is economical, since rewriting the programs would cost more than using the emulator.

Table 1.16

The evolution of database storage devices and input/output devices used in transaction processing systems The point is that most of the change has been in the area of input/output devices. If programs are to have a lifetime of more than a few years, they must have both database data independence and terminal device independence.

Year	Database Devices	Input/Output Devices
1960	Cards, Tape, Disc	Cards/Listings
1970	Disc	Keyboard/Teletype
1980	Disc	Keyboard/Character-mode CRT
1990	Disc	Keyboard/Mouse/Bitmap display

In contrast to input/output devices, data storage has not changed much when compared to device interfaces. The rapid evolution and diversification of user interfaces shows every sign of continuing. The trend for data storage is for the database to move away from slow disc-based access to random-access memory. Programs written today must be insulated from such technology changes.

To insulate applications from technology changes in terminals, the classic data communications system presents an abstract, record-oriented interface to the application and presents a device-specific, formatted data stream to the device. This is a great convenience. For example, the programming language COBOL excels at moving records about. The COBOL program simply reads a record from the terminal, looks at it, and then inserts it in the database; or perhaps it uses the record as a search condition on the database and moves the resulting records to the terminal. This logic is shown inFigure 1.17.

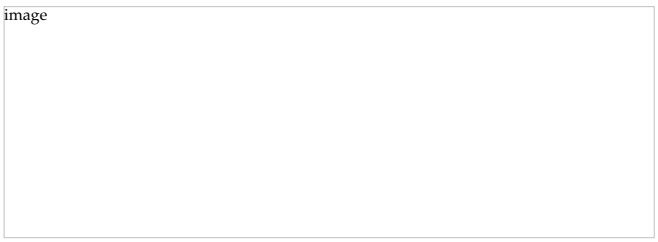


FIGURE 1.17 **The structure of an application running on a classic transaction processing system.** The DB-DC system provides a record-oriented interface to both the database (disc) and the network (terminals). The application just moves records from place to place. The details of terminal formats are left to the data communications system.

Presentation services provide device independence by using the technique diagrammed inFigure 1.18. First, an interactive *screen painter* produces a form description. The designer specifies how the screen will look by painting it. The screen painter translates this painting into an abstract *form description*, which can then be compiled to work for many different kinds of devices (e.g., each of the many character-mode displays and each of the many window systems). The description specifies how the form looks on the screen in abstract terms. In painting the screen, the designer says things like "this is a heading," "this is an input field," "this is an output field," "this is a decoration," and so on. Integrity checks can be placed on input fields. Ideally, all this is prespecified, since many inputs are database domains with well-defined types and integrity constraints. For example, the repository definitions of the part-number domain and customer name domain immediately imply a complete set of display attributes, help text, and integrity checks. Once the form is defined, it implies two records used to communicate with the application program: (1) an input record consisting of the input fields and their types, and (2) an output record consisting of the output fields and their types. The form may have function keys, menus, buttons, or other attributes that also appear as part of the input record.

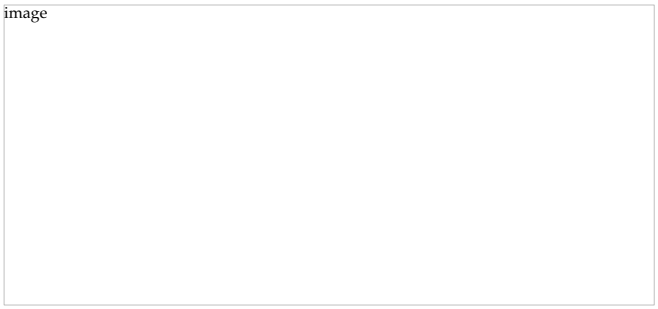


FIGURE 1.18 **How presentation management gives device independence.** The forms painter generates an abstract description of the form. The input and output fields of the form imply an input-output record structure for the program. The abstract form, when combined with a particular device description, can be used to read and write the device. The repository stores the forms descriptions, the record descriptions, and the device descriptions.

When the application reads a form, the presentation manager displays the form, gathers the input data, checks it against the form's integrity constraints, and then presents the form's input data as a record to the application. Conversely, when the application writes a record to the form, the presentation manager takes the form description, the device description, and the application data fields, and builds a device-specific image. In this way, the same program can read and write a teletype, a 24-by-80 character-mode terminal, a window of an X-Windows terminal, or a window of a PC running Microsoft Windows. That is device independence.

Along with the core service of device independence, the presentation manager also provides the screen painter and a screen debugging facility. It manages the form, screen, device, and record objects in the repository.

The presentation manager is the largest and most visible part of the data communications system. It often constitutes the bulk of the transaction processing system. For example, adding support for a *single* device feature—color displays on the 3270 terminal—involved adding 15,000 lines of code to CICS. This was more than the code needed to add distributed transactions to CICS. In general, data communications is bigger than the database system, TP monitor, and repository combined.

1.3.4.2 Client-Server Data Communications

Modern transaction processing systems are moving the bulk of presentation management to the client. The client interfaces to the TP monitor by issuing transactional RPCs. In this world, presentation services are an integral part of the client system.

When presentation services move to the client, the server has little control or even concern about how the data is gathered or displayed. The transaction processing monitor at the server receives a remote procedure call from the client. Moving presentation services funetions to the workstation is a vast simplification because the workstation has only a single device type, commonly a window.

System administration still presents the problem of many different clients, each with its own environment (e.g., DOS, OS/2, UNIX, Macintosh), its own presentation services, (e.g., PresentationManager, X-Windows, Motif, OPENLOOK, NextStep, Macintosh), and its own look and feel. The administrative problem, then, is still a nightmare, because each terminal has now been replaced with a complex workstation complete with local database, operating system, and networking system. If the system is centrally administered, the administrators can simplify the number of cases by dictating a *supported* set of environments.

In this client-server computation model, servers export transactional services, and workstation clients can pick any presentation format they like. The main job of the client-server data communication system is to provide transactional RPCs to these clients and servers. For an illustration of this situation, refer to Figure 1.6.

1.3.5 Database Features[4]

Database systems store and retrieve massive amounts of structured data. They provide a mechanism to *define* data structures that represent objects, and they provide a non-procedural interface to *manipulate* (read and write) that data. The interface is non-procedural in the sense that the program or person accessing the data need not know the location of the data (local, remote, main memory, disc, archive) or the detailed access paths used to find the data. Rather, the application requests operations on the abstract records or objects, while the database is responsible for mapping such abstractions to the concrete hardware. The definition of the data includes assertions about the data values and relationships. The database system enforces these assertions when the data is updated or when the transaction commits. Each database system acts as a resource manager and thereby provides ACID operations on its data. Each database system also provides utilities to manage and *control* the use of the data.

Because databases tend to last for decades, it is important that the representation and programs be adaptable to change. For a start, this requires that the database be an international standard (e.g., CODASYL or SQL), so that it can be moved to new equipment when the current computers are replaced with different models from different vendors. The database must insulate the applications from the location of the data and from the exact representation of the data. These ideas—*location transparency* and *data independence*—are very similar to the device independence issue discussed in Subsection 1.3.4.1.

To ease programming and operations, the database system should be integrated with a repository and with one or more application generators. These application generators give a visual interface to database definition, manipulation, and control. They automate much of the design process.

The database system should allow the designer to place parts of the database at different sites. As a rule, data is placed on computers near the data source or the data consumers (or both). This creates *partitioned* data—some parts of the table are here and some parts are there. If the same data is placed at many sites, it is said to be *replicated*. The repository gave a good example of this. For the sake of local autonomy, parts of the repository are replicated at each site. Some other parts of the repository are partitioned, since only the local node needs to know about local objects. Database systems providing transparent access to replicated and partitioned data are called *distributed database systems*.

1.3.5.1 Data Definition

SQL defines a standard set of *atomic types*[5] (numbers, character strings, timestamps, users, and so on). User-defined types, called *domains*, can be defined in terms of atomic types and other domains. For example, part number or customer name can be defined as domains. Each domain has a name (e.g., partno) along with constraints on the domain values (e.g., birthyear > 1850), formatting information (e.g., display as Kanji with heading), and comment information. These domain definitions are registered in the repository (the SQL catalogs describing all SQL tables are part of the repository) and can then be used to define fields in tables, screens, reports, and program variables.

SQL tables are sets of records, each record being a sequence of values. The distinguishing feature of a relational database is that all the records of a table have the same format (the same sequence of types or domains). This uniformity means that each record of the table has exactly the same form. Consequently, set-oriented operations like SORT, PROJECT, and SELECT apply to all records of a table in a natural way. Just as important, the result of each of these relational operators is again a table (relation); thus, the operators can be composed arbitrarily to form complex queries and operations on the database.

Since each operation produces a table as output, it is possible to define virtual tables, called *views*, that are computed from one or more underlying tables. Views provide data independence: If a table definition is altered, the old table format can often be defined as a view of the new table format, and old programs can operate unchanged by using the view to access the table (as in Figure 1.19).

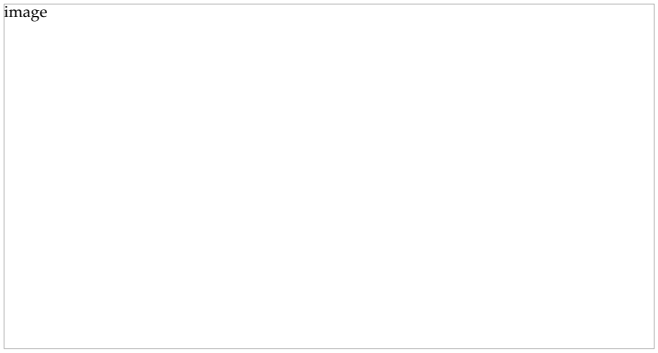
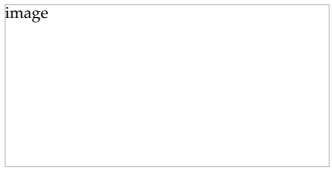


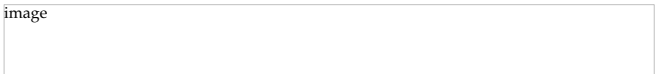
FIGURE 1.19 The basic concepts of logical data definition in a relational database system.

Tables can constrain the values allowed in them. Table columns inherit the constraints of their constituent domains. The table definition can specify that some column values are *unique*; that is, no other record in the table can have this value. The next step is to specify that some set of field values must occur in some other table; for example, invoice part numbers must appear in the parts table. Such rules are called *referential integrity* constraints. Inserts, updates, and deletes will be rejected if they violate these constraints. The generalization of all these ideas is to invoke a procedure each time a record is read or written. The procedure can reject or accept the update and, in fact, may have side effects causing other records to be updated, inserted, or deleted. In their general form, such *trigger* procedures allow arbitrary views to be updated. They can translate a view update into a particular sequence of updates on the underlying tables, thereby resolving the ambiguity inherent in general view updates.

The general syntax of SQL data-definition operations is shown below:



Here is a simple SQL table definition showing the use of column constraints, referential integrity constraints, and uniqueness constraints:



The SQL standard covers the *logical data definition* issues. There are also *physical data definition* issues: where to place the data and how to organize it. The table needs to be stored somewhere. The database system has a pool of storage, and, if not instructed otherwise, it will automatically place the table in that pool and organize the records as a heap or sequential file. The database tries to pick reasonable defaults for the physical file attributes. Some of the file attributes that users often want to specify include organization (how the records are clustered—sequential, keyed, hashed), access paths (primary and secondary indices on the data to speed lookups), location (if the data is to be replicated or partitioned, the criteria must be specified), space allocation (block and extent size), and so on. These features are not standard, but vary from system to system. An extension of the previous example using Tandem's NonStop SQL syntax shows about one-fifth of these options. It partitions the table among two nodes and specifies the blocksize and extentsize:

All this logical and physical information is registered by the SQL system in sets of tables, called the SQL catalogs, within the repository. If a particular table is partitioned or replicated at many sites, then, for the sake of node autonomy, the table's definition is replicated in catalogs at each of those sites.

1.3.5.2 Data Manipulation

A key principle of the relational model is the use of one language for data definition, data manipulation, and data retrieval. In addition, the same language is used for both interactive queries and for programmatic access. This vastly reduces documentation and education—you only have to learn one language, then apply it in many contexts. Using the same language for a programmatic and conversational interface also provides a useful way to test queries: You can interactively enter your program and see the answer. The key concept that makes this work is that each operator takes relations as parameters, and a read operation returns a relation as its result. Relations have a natural display format and can be fed to further operators. In particular, views can be defined in terms of queries, as in Figure 1.19. The data-definition language borrows the expression handling and procedures from the data- (see Table 1.16) language to define constraints and triggers. The SQL SELECT operator forms the core of this logic. It is composed of three common operators: PROJECT, which discards some columns, SELECT, which discards some rows based on a predicate, and JOIN, which combines records from two tables to form a third table (see Figure 1.20).

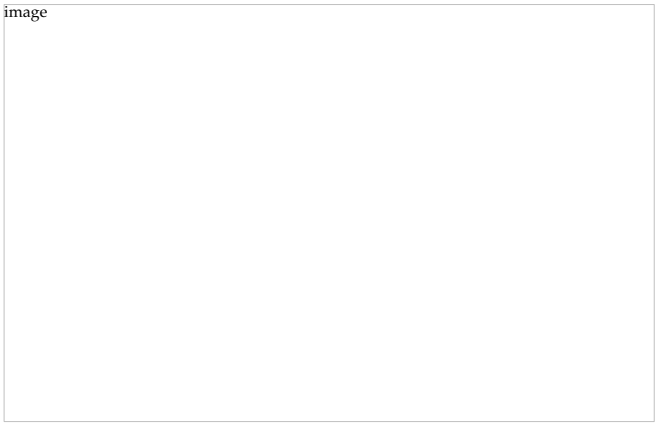


FIGURE 1.20 **The three basic relational operators: project, select, and join.** Each relational operator produces a new relation (table) based on one or more input tables.

The basic theorem of relational databases says that these operators are *Gödel complete*: They are equivalent to first-order predicate calculus [Codd 1971]. But the relational calculus is not *Turing complete*: It cannot describe all computable functions. A Turing-complete computational model is needed to write general programs. There is no need to invent such a language; almost any programing language will do. Consequently, SQL is always combined with a programming language. COBOL and C are the most popular ones today and create the languages CobSQL and CSQL.

The basic style of SQL, then, can be characterized as follows:

(1) Use the SELECT statement to define sets of records.

(2) Apply set-oriented INSERTS, UPDATES, and DELETEs where possible.

(3) Use a conventional programming language to manipulate sets when the database language is inadequate.

The connection between the host language and SQL has been called an *impedance mismatch* because SQL works on *sets of records* while the host language works on *individual records*. In addition, the SQL data types do not exactly match the host language data types. The way around the set-of-records problem is the same technique used to deal with files: The program deals with the set one record at a time. The program defines the set by a select statement and defines a *cursor* to address the current position in the set. The program then opens the cursor to allow it to fetch the next record, moves the cursor forward or backward in the set, and updates or deletes the cursor's current record (see Figure 1.21). The datatype mismatch is handled by coercion (conversions between the SQL data types and the host language types).

FIGURE 1.21 **Record-at-a-time programs deal with a set-oriented system like SQL by defining cursors on the set and then operating on one record at a time.** This is analogous to the way programs operate on sets of records in files.

1.3.5.3 Data Control

Data control is a catch-all. It includes security verbs (GRANT, REVOKE), concurrency control verbs (LOCK, SET TRANSACTION), and transaction verbs (BEGIN WORK, COMMIT WORK).

The basic security model of SQL is that tables are owned by users. Programs run as agents for users. SQL systems allow the owner to grant and revoke access to the tables he owns. By using views, the owner can grant others the authority to access a value-specific subset of the database. SQL's elaborate security model evolved from a timesharing system in which users share files. This model is inappropriate for a transaction processing system in which clients invoke servers that in turn access the data. If you think of the mail system example again, the servers at a node implement the mail database for all clients as a single set of tables. The servers typically run under the authority of the mail system and have access to all the tables. The servers authorize the clients to services on particular mailboxes. The mail system uses SQL security to completely hide the data from others; it encapsulates the data. Users can access the data only by invoking mail procedures, which in turn access the mail database. Security is a systems issue, and the database security must be integrated with the rest of the system.

Similar comments apply to transaction verbs. Historically, the database was the only transactional resource manager; it therefore implemented transaction verbs. In fact, `commit work` and `rollback work` are SQL verbs. As other transactional systems appear (such as persistent programming languages and transactional user interfaces), the transaction verbs become a system-wide facility, and the database system is just a part of the picture. It must operate as a resource manager within the context of the transaction processing system. The X/Open standards body is in the process of defining such system-wide transaction verbs (see Section 16.6).

In summary, security and transaction control are systems issues. The database must interoperate with the security and transaction mechanisms of the host operating system and the host transaction processing system.

1.3.5.4 Data Display

Once tables have been defined and populated with data, it is essential to have tools to browse the data and generate ad hoc reports. Such tools generally support a batch-oriented report writer that, given an SQL query and a report layout, produces a file or listing containing the answer. The report may consist of tables, charts, and graphs, or the output file may be fed to a tool like Lotus or Excel for data analysis and presentation.

With the move to online databases, these batch reports increasingly are being replaced by online reports in which an input form specifies the parameters to the query, and the output form specifies the answer. Again the output may be tables, reports, or graphs.

Virtually all systems come with a database browser that produces a form-per-record or a tabular display (see Figure 1.22). Such browsers let users query and edit databases much as one would query and edit text files. These displays can automatically be generated from the descriptive information about the table in the repository.

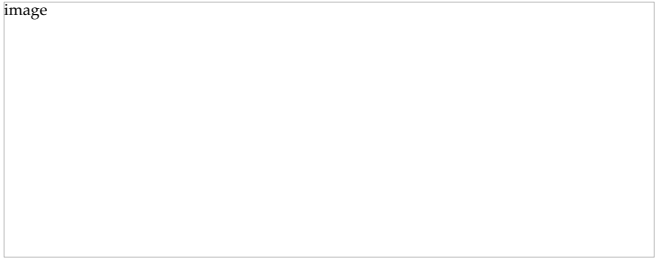


FIGURE 1.22 Database browser screens in two formats are automatically generated for each table from the information in the repository (`sql` catalogs).

1.3.5.5 Database Operations Utilities

Database utilities provide the operations tools needed to manage the database. Increasingly, these utilities run automatically, performing database archiving, recovery, reorganization, and redesign. Performance is usually a major issue for utilities. For example, if loading or archiving runs at one megabyte/second, then loading or dumping a terabyte database will take a million seconds, or about 12 days. There are no easy answers to this problem. Historically, utilities required exclusive access to the database; all other activities were suspended. This standalone approach is not acceptable for systems that must be continuously available for online transactions. The requirements are clear: (1) all utilities should run in the background without disrupting service, and (2) no utility should take more than a

day to do its job.

At present, there is considerable activity to make all database utilities *online* (operates on the data while others are reading and writing it), *incremental* (works on a few megabytes at a time), *restartable* (continues if stopped due to a fault or overload), and *parallel* (operates on multiple processors and disks in parallel to speed up the operation). Such utilities now exist for loading, dumping, recovering, and reorganizing databases (e.g., reclustering B-trees), and for collecting garbage and compacting disks. We know algorithms to build indices while the system is operating and the base table is changing, but no one has incorporated these algorithms into production systems yet. Certain operations, like dropping a table's column, invalidate the table. Such operations may force the application to be recompiled or even reprogrammed. There are no proposed solutions for such problems.

Other utilities produce either batch or interactive reports on the performance of the system. At present, the typical style is to provide a report telling which programs access which tables. These reports are usually broken down by table or by statement within program. Associated with each entry is a processor cost, a disk I/O cost, and, if the system is distributed, a message cost. This information can be used to analyze system performance.

Tools are emerging that analyze performance information and suggest better physical designs. In the future, the tools will be integrated with the utilities to make the system self-tuning.

1.3.6 Operations Features

As explained in Subsection 1.2.2, system administration and operations are a major part of the cost of owning a transaction processing system. The routine and mundane should be automated, both to reduce staff costs and to reduce errors. Humans should only be asked to deal with exceptional situations. Database archiving and reorganization are examples of such routine tasks.

There should be a forms-oriented interface to display the system status and performance at several levels of abstraction (e.g., application, process, file, or device). These displays should include mechanisms to detect problems, diagnose them, and suggest remedies. The system should also track problems (e.g., broken hardware and software) and generate reports on the status of pending problems. The repository is the key mechanism for storing all this information. It is also the basis for writing operations applications using high-level tools.

The system provides the mechanism for security, recovery, and change control, but the operations and administration staff must design the policy, specifying how these mechanisms will be used. For example, there needs to be a policy on how to change users, devices, applications, and system software.

Once the policies are in place, the administration and operations staff should conduct frequent fire drills and audits to assure that the policies work and are actually being followed.

1.3.7 Education and Testing Features

Education is an important area where we have no sage advice—only the following platitudes. It is important for the transaction processing system to use a standard interface (such as SQL, COBOL, or PresentationManager), and a standard look and feel (such as Motif, OpenLook, or Macintosh), so that users do not have to learn new skills. Programmer and user-skill portability is probably more important than program portability. This is the main theme of IBM's System Application Architecture [IBM-SAA]. Use of such standards reduces the need for education and training. That, in turn, has substantial benefits for the cost of system ownership.

Education on how to operate or use the system can be provided as *imbedded education*. This can take the form of help text for each function and a training mode in which operations against a toy version of the database are rolled back at the end of the operation.

Again, the imbedded education comes from the repository. The documentation of all functions and the meaning of all fields of each screen should be recorded in the repository. Likewise, the documentation on each entity, its possible values, referential integrity constraints, and diagnostic messages should all be recorded in the repository (in the different national languages). If this is done, help text for using and operating the system can be extracted automatically from the repository. Of course, the text will have to be manually translated to the user's native language (German, Korean, etc.).

The system must provide utilities to generate sample databases for testing. It must also generate and run multi-user input scripts to test the system for correctness (regression testing) and for performance under high load (stress testing).

1.3.8 Feature Summary

The transaction processing system includes one or more application generators built around a system repository. The repository is a database describing the entire system design and implementation. It is the mechanism whereby the administrator can assess and control change; its tools can automate—or at least guide—the administrator, operator, and designers through complex tasks.

The transaction processing system also includes a TP monitor that provides the core services of authorizing, scheduling, and executing transactions and resource managers. It provides the mechanisms to coordinate transaction commitment and isolation, along with an execution environment for resource managers. It coordinates their startup and checkpointing, as well as recovery of objects from the archive.

The transaction processing system has two key resource managers: database (DB) and data communications (DC). Thus, transaction processing systems are sometimes called DB-DC systems. The database is typically an SQL database system with data definition, data control, and data-manipulation verbs. The objects implemented by the database—tables, views, statements, and cursors—are transactional. The data communications component provides the basis for a transactional RPC mechanism. In the past, the data communications component also provided presentation services for dumb terminals, giving the application program a record-oriented, terminal-independent interface that is a major component of old transaction processing systems. As systems adopt client-server architectures, presentation services are being moved to the client and appear to be part of the workstation operating environment. This relieves the data communications system of these chores.

1.4 Summary

Transaction processing is thousands of years old; today, it is the dominant application of general-purpose computers. This chapter took a particular transaction processing application, electronic mail, and viewed it from the perspectives of those who use or operate it. A feature list describing the various components of a complete transaction processing system was presented. Underlying this chapter is the theme that transaction processing systems are in rapid evolution.

Traditional transaction processing systems are increasingly presented with many resource managers: databases, persistent programming languages, and type-specific resource managers. This diversity is forcing systems into a more open architecture than the classic DB-DC model. The movement of presentation services to clients, along with the increasing interconnection of transaction processing systems, is forcing a fully distributed design.

The main theme of this chapter is that the transaction concept is emerging as a paradigm for handling exceptions. Exceptions become increasingly important as client-server LAN-based systems evolve from simple disk and file servers to systems supporting distributed computation. When one member of the computation faults, the rest of the computation can be aborted, and a consistent system state can be reconstructed. Transactional RPC is being added both to the traditional transaction processing systems and to distributed operating systems, thus blurring the distinction between them.

A second theme of this chapter is that a transaction processing system is a large web of application generators, systems design and operations tools, and the more mundane language, database, network, and operations software. The repository and the applications that maintain it are the mechanisms needed to manage the TP system. The repository is a transaction-processing application. It represents the system configuration as a database and supplies change control by providing transactions that manipulate the configuration and the repository as a single transaction.

The transaction concept, like contract law, is intended to resolve the situation when exceptions arise. The first order of business in designing a system is, therefore, to have a clear model of system failure modes. What breaks? How often do things break? That is the topic of Chapter 3, which gives an empirical and abstract model of computer faults and computer fault tolerance. Once that is out of the way, the rest of the book develops the concepts and techniques used in transaction processing systems.

1.5 Historical Notes

National Cash Register (NCR, now part of AT&T) likes to point out that it was automating transaction processing at the turn of the century. Batch transaction processing using cards began in the 1890s, and it was in full swing by 1940. The first online transaction processing systems using teletypes, computers, and magnetic storage were built in about 1960; they were contemporary with early time-sharing systems.

By the late 1960s, database systems had been recognized as generic utilities, and the techniques to implement them for batch processing were well developed. Notably, the CODOL committee had a task group to define a standard database language (DBTG). The database community went on to evolve the relational model that was prototyped in the 1970s and standardized as SQL in 1987. Today, most database systems are SQL-like.

While this was happening, terminal monitors gradually evolved to support remote procedure calls. The most successful of these was (and still is) IBM's Customer Information and Control System (CICS). In the early 1970s, a typical CICS system was supporting 400 terminals on a processor with .5 MIPS, 256 KB of memory, and four disks of 20 MB each. Each input message would invoke a program to service it. Programs were usually short and simple, using the CICS file management system and other services. Systems routinely ran four such transactions per second. By the late 1970s, most database vendors had discovered CICS and were attaching their database products to it as resource managers. This produced a DB-DC system that could be used for transaction processing. This whole experience pointed out the need for a clean interface to resource managers.

Several other significant transaction processing systems evolved in that era. A reservation system, originally built by American Airlines, became an IBM product called SABRE. It was subsequently renamed Airlines Control Program (ACP), and then again renamed Transaction Processing Facility (TPF). TPF is designed for speed. Until quite recently, it was programmed only in assembly language, had only three record sizes (=400 bytes, =1000 bytes, and =4000 bytes), and all programs had to fit in a single record. TPF has few of the amenities mentioned in this chapter, but it is running most airlines reservations systems today and has the highest transaction throughput rates. It is fair to say that TPF trades people cost for computer cost, economizing on hardware. That was essential thirty years ago, but it is not a good trade-off today.

Both TPF and CICS were surprises. IMS was intended to be the real DB-DC system. CICS was only a TP monitor for uniprocessors, and TPF was too hard to use. By contrast, IMS had most of the niceties described here, and it had them very early (about 1973). It had ACID transactions, device independence, and data independence. IMS presented the application with the same hierarchical data model for both DB and DC, but IMS was late to exploit relational databases, TRPC, and intelligent networks. Gradually, CICS outnumbered it, and now CICS, combined with IBM's DB2 relational database, is the typical system being built. Today there are about 30,000 CICS systems and about 8,000 IMS systems. By that measure, IBM dominates the high end of the transaction marketplace. When one considers small systems each supporting tens of terminals, IBM's AS400 and CICS are the dominant products.

During the 1980s, several new transaction processing systems emerged, notably Encompass from Tandem and ACMS from Digital. These systems evolved rapidly and, due to their late start, have a more modern look. They are popular for distributed transaction processing applications.

Relational databases evolved parallel with transaction processing systems. At first they operated with one or two processes per terminal and, therefore, could not be scaled to very large terminal networks. Gradually they rediscovered the TP monitor architecture, in which a few servers perform work for many clients. Today, many database systems execute as servers in a host operating system. They provide stored procedures that, when invoked by clients, are executed by the database system as services. This is a form of transactional remote procedure call.

Fundamental changes are taking place today. One trend is the integration of transaction processing functionality with the operating system. For example, IBM's MVS system has a subsystem interface, a standard transactional local and remote procedure call (MVS/APPC), an integrated transaction manager, a log manager, and an integrated lock manager (IRLM). DEC's VMS has a transaction manager as a standard part of the operating system (DECdtm), and VMS provides a generic lock facility. The oldest such system, Tandem's Guardian system, has an integrated log and transaction manager (TMF), a transactional RPC (the message system and Pathway), and the database lock manager is a generic lock manager. Standard transaction protocols are being defined by IBM, ISO, and X/Open. Transactional RPC, then, is becoming ubiquitous.

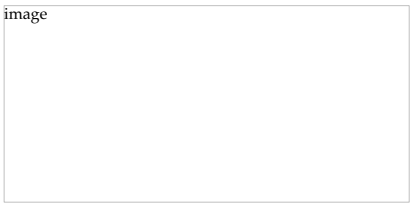
A second major trend is the emergence of new transaction processing systems, typically on UNIX systems. Digital is introducing ACMS, IBM is introducing CICS, NCR is introducing Topend, and Transarc is introducing Encina. All these are new or reimplemented TP systems joining the Tuxedo TP system, which has long run on UNIX. Each of these systems is designed or redesigned to fit the X/Open definition of distributed transaction processing (see Section 2.7 or 16.6). This trend to open transaction processing systems, and the trend to have a standard transactional remote procedure call mechanism, will be an enabling technology for distributed applications.

Exercises

1. [1.2, 10] You have an interest-bearing checking account at some bank. The standard transaction is to write a check debiting the account or to deposit funds in the account—the so-called DebitCredit transaction. Describe four scenarios—one for each of the four acid properties—including one that violates one property but does not violate the other three.
2. [1.2, 10] Describe eight different transaction programs on the checking account.
3. [1.3, 20] Find a *complete* manual set for some TP system (e.g., the IBM manual set for CICS and related products is a wall 2 meters high and 3 meters long, giving about 18 linear meters of manuals). Measure the linear feet of manuals in the following areas: hardware, operating system, database, networks, languages (e.g., COBOL), tools (edit, debug, bind, make), operator and administrator utilities, application generators, and applications. Draw a pie chart of the result. This pie chart approximately reflects the relative size of each component.
4. [1.3, project] Find a TP system and a person in each role (user, administrator, designer). Interview each person about how the system works and what he likes and dislikes about it. Write a ten-page description of the system from each perspective.
5. [1.3, project] Find a modern application generator that supports windows, graphics, and rapid prototyping. Implement the user interface shown in Figure 1.2. Describe the resulting database design in SQL. Produce a diagram of the application generator's repository/dictionary. Now imagine that the database is on the host. Describe the services on the host and their input and output parameters.
6. [1.2, 15] There are some interesting consequences of Figures 1.2 and 1.3. What is the information flow if a European user visits Australia and wants to read his mail while in Perth by connecting to the Hong Kong regional center?
7. [1.3.2, 15] List 20 objects that ought to be recorded in a complete repository.

Answers

1. A: The check gets lost in mail and so the bank database is not updated. C: You write a check for more than the current account balance. I: The bank pays you interest after you write a check but before the check is cashed. D: You lose your checkbook and balance.
2. Create Account, DebitCredit, CloseAccount, PayInterest, MonthlyStatement, YearlyStatement, Query, CancelCheck. Explain the function of each of these.
3. For the Tandem System: There are 56 linear feet of software manuals from the vendor (and 23 feet of hardware maintenance manuals). Third-party applications are about 200 additional feet. Ignoring the hardware and applications manuals the pie chart is:



4. Hint: Break into groups of two or three to do this project. Contact a hospital, bank, retailer, manufacturer, library, or phone company. Treat all information you receive as confidential, and have the interviewee review the report before distribution. This exercise should go on in parallel with the class. It is very educational.
5. Hint: Look at Paradox on a pc or 4th Dimension on a Macintosh.
6. Presumably, the workstation client sends his request to the local host that happens to be in Hong Kong (see Figure 1.3). The Hong Kong system could either forward the request (rpc) to the home mail system of this European user (Berlin), or it could access the Berlin mailbox directly from Hong Kong, using the distributed database facilities. It is likely that for manageability and performance reasons, the logon and subsequent services will be performed for the client by a Berlin server rather than the Hong Kong server. This logic is application specific and is part of the application design.

[1]The term ACID was coined by Haërder and Reuter [1983]. The term *serializable* is a commonly used synonym for isolated. The terms *persistent* and *stable* are widely used synonyms for durable. None of these alternatives (e.g., PACS, SACS) gives a nice acronym like ACID.

[2]As the name suggests, the bill-of-materials for an item is a list of assemblies used in the item. These assemblies may have subassemblies, so the bill-of-materials is actually a tree of dependencies. For example, a computer has a processor, memory, network interface, and power supply. Each of these components in turn has a bill of materials.

[3]God must love standards: He made so many of them.

[4]We assume that the reader already knows C and SQL. If that is not the case, then there are many good books on the topics. For SQL in particular, *An Introduction to DB2* [Date and White 1989] is a very readable tutorial. The following whirlwind tour presumably recapitulates what you already know from having read such a book. This book takes a dogmatic SQL view of databases, ignoring the "old" network proposals and the "new" object-oriented proposals. We are enthusiastic about the object-oriented ideas, but this is not the place to explore them. Sticking to SQL simplifies the presentation. Most of the transaction concepts and techniques used for relational databases will be needed in object-oriented databases as well.

[5]The term *atomic type* should not be confused with transaction atomicity. Programming language type systems are built from *basic types* (atomic types) and *constructors*. The basic types are character, integer, real, pointer, and so on. Records, vectors, and lists are constructors. Transaction atomicity is a completely independent concept.