CHAPTER

Introduction

1

1.1 THE BASICS

The Problem

A **business transaction** is an interaction in the real world, usually between an enterprise and a person or another enterprise, where something is exchanged. For example, it could involve exchanging money, products, information, or service requests. Usually some bookkeeping is required to record what happened. Often this bookkeeping is done by a computer, for better scalability, reliability, and cost. Communications between the parties involved in the business transaction is often done over a computer network, such as the Internet. This is **transaction processing (TP)**—the processing of business transactions by computers connected by computer networks. There are many requirements on computer-based transaction processing, such as the following:

- A business transaction requires the execution of multiple operations. For example, consider the purchase of an item from an on-line catalog. One operation records the payment and another operation records the commitment to ship the item to the customer. It is easy to imagine a simple program that would do this work. However, when scalability, reliability, and cost enter the picture, things can quickly get very complicated.
- Transaction volume and database size adds complexity and undermines efficiency. We've all had the experience of being delayed because a sales person is waiting for a cash register terminal to respond or because it takes too long to download a web page. Yet companies want to serve their customers quickly and with the least cost.
- To scale up a system for high performance, transactions must execute concurrently. Uncontrolled concurrent transactions can generate wrong answers. At a rock concert, when dozens of operations are competing to reserve the same remaining seats, it's important that only one customer is assigned to each seat. Fairness is also an issue. For example, Amazon.com spent considerable effort to ensure that when its first thousand Xboxes went on sale, each of the 50,000 customers who were vying for an Xbox had a fair chance to get one.
- If a transaction runs, it must run in its entirety. In a retail sale, the item should either be exchanged for money or not sold at all. When failures occur, as they inevitably do, it's important to avoid partially completed work, such as accepting payment and not shipping the item, or vice versa. This would make the customer or the business very unhappy.

- Each transaction should either return an acknowledgment that it executed or return a negative acknowledgment that it did not execute. Those acknowledgments are important. If no acknowledgment arrives, the user doesn't know whether to resubmit a request to run the transaction again.
- The system should be incrementally scalable. When a business grows, it must increase its capacity for running transactions, preferably by making an incremental purchase—not by replacing its current machine by a bigger one or, worse yet, by rebuilding the application to handle the increased workload.
- When an electronic commerce (e-commerce) web site stops working, the retail enterprise is closed for business. Systems that run transactions are often "mission critical" to the business activity they support. They should hardly ever be down.
- Records of transactions, once completed, must be permanent and authoritative. This is often a legal requirement, as in financial transactions. Transactions must never be lost.
- The system must be able to operate well in a geographically distributed environment. Often, this implies that the system itself is distributed, with machines at multiple locations. Sometimes, this is due to a legal requirement that the system must operate in the country where the business is performed. Other times, distributed processing is used to meet technical requirements, such as efficiency, incremental scalability, and resistance to failures (using backup systems).
- The system should be able to personalize each user's on-line experience based on past usage patterns. For a retail customer, it should identify relevant discounts and advertisements and offer products customized to that user.
- The system must be able to scale up predictably and inexpensively to handle Internet loads of millions of potential users. There is no way to control how many users log in at the same time or which transactions they may choose to access.
- The system should be easy to manage. Otherwise, the system management staff required to operate a large-scale system can become too large and hence too costly. Complex system management also increases the chance of errors and hence downtime, which in turn causes human costs such as increased stress and unscheduled nighttime work.

In summary, transaction processing systems have to handle high volumes efficiently, avoid errors due to concurrent operation, avoid producing partial results, grow incrementally, avoid downtime, never lose results, offer geographical distribution, be customizable, scale up gracefully, and be easy to manage. It's a tall order. This book describes how it's done. It explains the underlying principles of automating business transactions, both for traditional businesses and over the Internet; explores the complexities of fundamental technologies, such as logging and locking; and surveys today's commercial transactional middleware products that provide features necessary for building TP applications.

What Is a Transaction?

An **on-line transaction** is the execution of a program that performs an administrative function by accessing a shared database, usually on behalf of an on-line user. Like many system definitions, this one is impressionistic and not meant to be exact in all its details. One detail is important: A transaction is always the *execution* of a program. The program contains the steps involved in the business transaction—for example, recording the sale of a book and reserving the item from inventory.

We'll use the words **transaction program** to mean the program whose execution is the transaction. Sometimes the word "transaction" is used to describe the message sent to a computer system to request the execution of a transaction, but we'll use different words for that: a **request message**. So a transaction always means the execution of a program.

We say that a transaction performs an administrative function, although that isn't always the case. For example, it could be a real-time function, such as making a call in a telephone switching system or controlling a machine tool in a factory process-control system. But usually there's money involved, such as selling a ticket or transferring money from one account to another.

Most transaction programs access shared data, but not all of them do. Some perform a pure communications function, such as forwarding a message from one system to another. Some perform a system administration function, such as resetting a device. An application in which no programs access shared data is not considered true transaction processing, because such an application does not require many of the special mechanisms that a TP system offers.

There is usually an on-line user, such as a home user at a web browser or a ticket agent at a ticketing device. But some systems have no user involved, such as a system recording messages from a satellite. Some transaction programs operate **off-line**, or in batch mode, which means that the multiple steps involved may take longer than a user is able to wait for the program's results to be returned—more than, say, ten seconds. For example, most of the work to sell you a product on-line happens after you've entered your order: a person or robot gets your order, picks it from a shelf, deletes it from inventory, prints a shipping label, packs it, and hands it off to the shipping company.

Transaction Processing Applications

A **transaction processing application** is a collection of transaction programs designed to do the functions necessary to automate a given business activity. The first on-line transaction processing application to receive widespread use was an airline reservation system: the SABRE system developed in the early 1960s as a joint venture between IBM and American Airlines. SABRE was one of the biggest computer system efforts undertaken by anyone at that time, and still is a very large TP system. SABRE was spun off from American Airlines and is now managed by a separate company, Sabre Holdings Corporation, which provides services to more than 200 airlines and thousands of travel agencies, and which runs the Travelocity web site. It can handle a large number of flights, allow passengers to reserve seats and order special meals months in advance, offer bonuses for frequent flyers, and schedule aircraft maintenance and other operational activities for airlines. Its peak performance has surpassed 20,000 messages per second.

Today, there are many other types of TP applications and new ones are emerging all the time. We summarize some of them in Figure 1.1. As the cost of running transactions and of managing large databases decreases, more types of administrative functions will be worth automating as TP applications, both to reduce the cost of administration and to generate revenue as a service to customers.

In its early years, the TP application market was driven primarily by large companies needing to support administrative functions for large numbers of customers. Such systems often involve thousands of terminals, dozens of disk drives, and many large processors, and can run hundreds of thousands of transactions per day. Large TP systems are becoming even more important due to the popularity of on-line services on the Internet. However, with the downsizing of systems has come the need for small TP applications too, ones with just a few browsers connected to a small server machine, to handle orders for a small catalog business, course registrations for a school, or patient visits to a dental office. All these applications—large and small—rely on the same underlying system structure and software abstractions.

Application	Example Transaction
Banking	Withdraw money from an account
Securities trading	Purchase 100 shares of stock
Insurance	Pay an insurance premium
Inventory control	Record the fulfillment of an order
Manufacturing	Log a step of an assembly process
Retail point-of-sale	Record a sale
Government	Register an automobile
Online shopping	Place an order using an on-line catalog
Transportation	Track a shipment
Telecommunications	Connect a telephone call
Military Command and Control	Fire a missile
Media	Grant permission to download a video

FIGURE 1.1

Transaction Processing Applications. Transaction processing covers most sectors of the economy.

TP systems also are being offered as services to other companies. For example, Amazon.com hosts other companies' web storefronts. Some airlines develop and operate reservation services for other airlines. Some vendors of packaged applications are now offering their application as a service that can be invoked by a third party's application over the Internet, which in turn helps the third party offer other TP services to their customers. Given the expense, expertise, and management attention required to build and run a high-quality TP system, this trend toward out-sourcing TP applications is likely to grow.

A Transaction Program's Main Functions

A transaction program generally does three things:

1. Gets input from a web browser or other kind of device, such as a bar-code reader or robot sensor.
2. Does the real work being requested.
3. Produces a response and, possibly, sends it back to the browser or device that provided the input.

Each invocation of the transaction program results in an independent unit of work that executes exactly once and produces permanent results. We'll have more to say about these properties of a transaction program shortly.

Most TP applications include some code that does not execute as a transaction. This other code executes as an ordinary program, not necessarily as an independent unit of work that executes exactly once and produces permanent results. We use the term TP application in this larger sense. It includes transaction programs, programs that gather input for transactions, and maintenance functions, such as deleting obsolete inventory records, reconfiguring the runtime system, and updating validation tables used for error-checking.

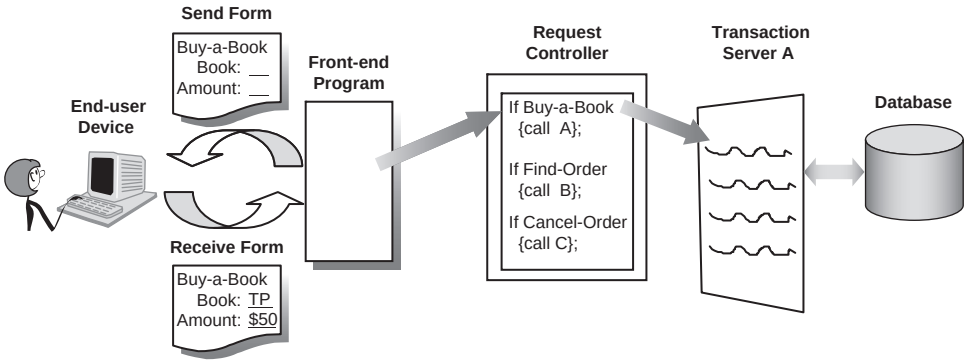

FIGURE 1.2

Transaction Application Parts. A transaction application gathers input, routes the input to a program that can execute the request, and then executes the appropriate transaction program.

1.2 TP SYSTEM ARCHITECTURE

A **TP system** is the computer system—both hardware and software—that hosts the transaction programs. The software parts of a TP system usually are structured in a special way. As you can see from Figure 1.2, the TP system has several main components. Different parts of the application execute in each of these components.

1. End-user device: An **end user** is someone who requests the execution of transactions, such as a customer of a bank or of an Internet retailer. An end-user device could be a physical device, such as a cash register or gasoline pump. Or it could be a web browser running on a desktop device, such as a personal computer (PC). If it is a dumb device, it simply displays data that is sent to it and sends data that the user types in. If it is a smart device, then it executes application code that is the front-end program.

2. Front-end program: A **front-end program** is an application code that interacts with the end-user device. Usually it sends and receives menus and forms, to offer the user a selection of transactions to run and to collect the user's input. Often, the device is a web browser and the front-end program is an application managed by a web server that communicates with the browser via HTTP. The front-end program validates the user's input and then sends a request message to another part of the system whose job is to actually execute the transaction.

3. Request controller: A **request controller** is responsible for receiving messages from front-end programs and turning each message into one or more calls to the proper transaction programs. In a centralized system, this is simply a matter of calling a local program. In a distributed TP system, it requires sending the message to a system where the program exists and can execute. If more than one program is needed, it tracks the state of the request as it moves between programs.

4. Transaction server: A **transaction server** is a process that runs the parts of the transaction program that perform the work the user requested, typically by reading and writing a shared database, possibly calling other programs, and possibly returning a reply that is routed back to the device that provided the input for the request.

5. Database system: A **database system** manages shared data that is needed by the application to do its job.

For example, in an Internet-based order processing application, a user submits orders via a web browser. The front-end program is managed by a web server, which reads and writes forms and menus and perhaps maintains a shopping cart. A request controller routes requests from the web server to the transaction server that can process the order the user requested. The transaction server processes the order, which requires accessing the database that keeps track of orders, catalog information, and warehouse inventory, and perhaps contacts another transaction server to bill a credit card for the order.

The transaction programs that run in the server are of a limited number of types that match operational business procedures, such as shipping an order or transferring funds. Typically there are a few dozen and usually no more than a few hundred. When applications become larger than this, usually they are partitioned into independent applications of smaller size. Each one of these programs generally does a small amount of work. There's no standard concept of an average size of a transaction program, because they all differ based on the application. But a typical transaction might have between zero and 30 disk accesses, a few thousand up to a few million instructions, and at least two messages, but often many more depending on how distributed it is. It may be distributed because different application services are needed to process it or because multiple machines are needed to handle the application load. The program generally is expected to execute within a second or two, so that the user can get a quick response. Later on we'll see another, more technical reason for keeping transactions short, having to do with locking conflicts.

Database systems play a big role in supporting transaction programs, often a bigger role than the application programs themselves. Although the database can be small enough to fit in main memory, it is often much larger than that. Some databases for TP require a large number of nonvolatile storage devices, such as magnetic or solid state disks, pushing both storage and database system software technology to the limit. To scale even larger, the database may be replicated or partitioned onto multiple machines.

Another major category of TP software products is **transactional middleware**, which is a layer of software components between TP applications and lower level components such as the operating system, database system, and system management tools. These components perform a variety of functions. They can help the application make the most efficient use of operating system processes, database connections, and communications sessions, to enable an application to scale up. For example, they may provide functions that client applications can use to route requests to the right server applications. They can integrate the transaction abstraction with the application, operating system, and database system, for example, to enable the execution of distributed transactions, sometimes across heterogeneous environments. They can integrate system management tools to simplify application management, for example, so that system managers can balance the load across multiple servers in a distributed system. And they may offer a programming interface and/or configurable properties that simplify the use of related services that originate in the operating system and database system.

Transactional middleware product categories have evolved rapidly over the past fifteen years. Before the advent of the World Wide Web (WWW), transactional middleware products were called TP monitors or on-line TP (OLTP) monitors. During the mid 1990s, application server products were introduced to help application developers cope with new problems introduced by the Web, such as integrating with web servers and web browsers. Initially, application servers formed a bridge between existing commercial systems managed by TP monitors and the Internet. In a relatively short time, the functionality of application servers and TP monitors converged. During the same period, message-oriented transactional middleware and object request brokers became popular. Message-oriented middleware became the foundation of a product category called enterprise application integration systems. The adoption of standard Internet-based protocols for application communication, called Web Services, has led to the enterprise service bus, another transactional middleware product. And finally, workflow products have become popular to help users define and manage long-running business processes. Although transactional middleware products usually are marketed as a complete environment for developing and executing TP applications, customers sometimes use components from multiple transactional middleware products to assemble their TP environments.

Service Oriented Computing

Service Oriented Architecture (SOA) is a style of design in which applications are composed in whole or in part of reusable services. SOA aligns information systems technology well with business objectives by modeling an application as a composition of reusable services. In contrast to the object-oriented (OO) paradigm, services are designed to model functions rather than things. They are a natural abstraction of the concept of business services; that is, services that a business provides to its customers and partners. A service can be implemented using an object, but it need not be. For example, it may be implemented using a procedure, stored procedure, asynchronous message queue, or script. Services are characterized by the messages they exchange and by the interface contracts defined between the service requester and provider, rather than by the programs that are used to implement them.

Service orientation has been around for a long time as a concept. However, only recently has it become mainstream, with many large-scale web sites for web search, social networking, and e-commerce now offering service-oriented access to their functions. In part, this wide availability is due to the advent of standard Web Services protocols. Web Services is an implementation technology that enables independent programs to invoke one another reliably and securely over a network, especially the Internet. Many vendors now support Web Services protocols. This enables one to implement SOA in a multivendor environment, which is a requirement for most enterprises.

A TP system that is created in whole or in part using the SOA approach may include multiple reusable services offered by a single transaction program or by multiple distributed services. An SOA-based TP system may include both synchronous and asynchronous communications mechanisms, depending on the message exchange patterns that a given service supports and the execution environment in which it runs. SOA-based TP systems may be assembled using a combination of services from a variety of applications and using a variety of operating systems, middleware platforms, and programming languages.

Figure 1.3 illustrates the components of a service-oriented architecture. They include a service provider that offers a service, a requester that invokes a service, and a registry (sometimes called a repository) that publishes service descriptions. The service descriptions typically include the service interface, the name and format of data to be exchanged, the communications protocol to be used, and the quality of service that the interaction is required to support (such as its security and reliability characteristics and its transaction behavior).

A caller communicates with a service by sending messages, guided by a message exchange pattern. The basic pattern is a one-way asynchronous request message, where a caller sends a request message to the service provider and the service provider receives the message and executes the requested service. Other common patterns are request-response and publish-subscribe.

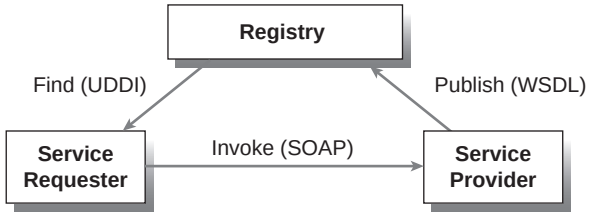

FIGURE 1.3

Basic Architecture of Service Orientation. A service provider publishes its interface in the registry. A service requester uses the registry to find a service provider and invokes it. The corresponding Web Service technologies are WSDL, UDDI, and SOAP.

The registry is an optional component because the requester can obtain service description information in other ways. For example, a developer who writes the requester can find the service description on a web site or be given the service description by the service's owner.

One mechanism to implement SOA is Web Services, where a service requester invokes a service provider using the protocol SOAP.[1] The service interface offered by the service provider is defined in the Web Services Description Language (WSDL). The service provider makes this interface known by publishing it in a registry. The registry offers access to service descriptions via the Universal Description, Discovery, and Integration (UDDI) protocol. A service requester and provider can be running in different execution environments, such as Java Enterprise Edition or Microsoft. NET.

Web Service interfaces are available for virtually all information technology product categories: application servers, object request brokers, message oriented middleware systems, database management systems, and packaged applications. Thus, they provide **interoperability**, meaning that applications running on disparate software systems can communicate with each other. Web Services support transaction interoperability too, as defined in the Web Services Transactions specifications (discussed in Section 10.8).

Services simplify the assembly of new applications from existing ones by combining services. Tools and techniques are emerging to simplify the assembly of services, such as the Service Component Architecture for Java and the Windows Communication Foundation for Windows.

A TP application may exist as a combination of reusable services. The use of reusable services doesn't change the functions of the front-end program, request controller, or transaction server. However, it may affect the way the functions are designed, modeled, and implemented. For example, in Figure 1.2, the decision to build the request controller as a reusable Web Service may affect the choice of implementation technologies, such as defining the interface to the request controller using WSDL and invoking it using SOAP. That decision may also affect the design by enabling an end-user device such as a web browser to call the request controller service(s) directly, bypassing the front-end program. We'll talk a lot more about TP software architecture in Chapter 3.

Representational State Transfer (REST) is another approach to SOA, rather different than that of Web Services. The term REST is used in two distinct but related ways: to denote the protocol infrastructure used for the World Wide Web, namely the Hypertext Transfer Protocol (HTTP); and to denote a software architectural pattern that can be implemented by web protocols. We use it here in the former sense, which we call REST/HTTP. We will discuss the REST architectural pattern in Section 3.3.

REST/HTTP focuses on the reuse of resources using a small set of generic HTTP operations, notably GET (i.e., read), PUT (i.e., update), POST (i.e., insert), and DELETE. This is in contrast to Web Services, which uses services that are customized for a particular application. Each HTTP operation is applied to a resource identified by a Uniform Resource Identifier (URI). A registry function, as shown in Figure 1.3, is needed to translate each URI into a network address where the resource can be found. On the Internet, this is implemented by the Domain Name System, which translates domain names such as www.mydomain.com into IP addresses.

In REST, generic HTTP operations are used to perform application-specific functions. For example, instead of invoking a Web Service AddCustomer, you could use REST to invoke the POST operation with a URI that makes it clear that a customer is being inserted, such as www.company-xyz.com/customers. In general, the application-specific information that identifies the function and its parameters must be embodied in the representation of the resource. This is why this style of communication is called representational state transfer. In practice, the representation that is transferred is often in a standard, stylized form, such as JavaScript Object Notation (JSON).

The format of the representation is specified in the HTTP header; the `content-type` and `accept` fields specify the format of the input and output, respectively. Thus, instead of specifying data types in a service's

[1]Originally, SOAP was an acronym for Simple Object Access Protocol. However, the SOAP 1.2 specification explicitly says it should no longer be treated as an acronym.

interface definition, the caller specifies the data types it would like to receive. This flexibility makes it easier for diverse kinds of callers to invoke the service.

REST/HTTP is popular for its speed and simplicity. Web Services require parameters in SOAP messages to be represented in XML, which is expensive to parse. XML is self-describing and highly interoperable, but these benefits are not always important, for example, for simple services. A very simple interface makes it easier and faster to manipulate in limited languages such as JavaScript.

Hardware Architecture

The computers that run these programs have a range of processing power. A display device could be a character-at-a-time terminal, a handheld device, a low-end PC, or a powerful workstation. Front-end programs, request controllers, transaction servers, and database systems could run on any kind of server machine, ranging from a low-end server machine, to a high-end multiprocessor mainframe, to a distributed system. A distributed system could consist of many computers, localized within a machine room or campus or geographically dispersed in a region or worldwide.

Some of these systems are quite small, such as a few display devices connected to a small machine on a PC Local Area Network (LAN). Big TP systems tend to be enterprise-wide or Internet-wide, such as airline and financial systems, Internet retailers, and auction sites. The big airline systems have on the order of 100,000 display devices (terminals, ticket printers, and boarding-pass printers) and thousands of disk drives, and execute thousands of transactions per second at their peak load. The biggest Internet systems have hundreds of millions of users, with tens of millions of them actively using the system at any one time.

Given this range of capabilities of computers that are used for TP, we need some terminology to distinguish among them. We use standard words for them, but in some cases with narrower meanings than is common in other contexts.

We define a **machine** to be a computer that is running a single operating system image. It could use a single-core or multicore processor, or it could be a shared-memory multiprocessor. Or it might be a virtual machine that is sharing the underlying hardware with other virtual machines. A **server machine** is a machine that executes programs on behalf of client programs that typically execute on other computers. A **system** is a set of one or more machines that work together to perform some function. For example, a **TP system** is a system that supports one or more TP applications. A **node** (of a network) is a system that is accessed by other machines as if it were one machine. It may consist of several machines, each with its own network address. However, the system as a whole also has a network address, which is usually how other machines access it.

A **server process** is an operating system process, *P*, that executes programs on behalf of client programs executing in other processes on the same or different machines as the one where *P* is running. We often use the word "server" instead of "server machine" or "server process" when the meaning is obvious from context.

1.3 ATOMICITY, CONSISTENCY, ISOLATION, AND DURABILITY

There are four critical properties of transactions that we need to understand at the outset:

- Atomicity: The transaction executes completely or not at all.
- Consistency: The transaction preserves the internal consistency of the database.
- Isolation: The transaction executes as if it were running alone, with no other transactions.
- Durability: The transaction's results will not be lost in a failure.

This leads to an entertaining acronym, ACID. People often say that a TP system executes ACID transactions, in which case the TP system has "passed the ACID test." Let's look at each of these properties in turn and examine how they relate to each other.

Atomicity

First, a transaction needs to be **atomic** (or **all-or-nothing**), meaning that it executes completely or not at all. There must not be any possibility that only part of a transaction program is executed.

For example, suppose we have a transaction program that moves $100 from account A to account B. It takes $100 out of account A and adds it to account B. When this runs as a transaction, it has to be atomic—either both or neither of the updates execute. It must not be possible for it to execute one of the updates and not the other.

The TP system guarantees atomicity through database mechanisms that track the execution of the transaction. If the transaction program should fail for some reason before it completes its work, the TP system will undo the effects of any updates that the transaction program has already done. Only if it gets to the very end and performs all of its updates will the TP system allow the updates to become a permanent part of the database.

If the TP system fails, then as part of its recovery actions it undoes the effects of all updates by all transactions that were executing at the time of the failure. This ensures the database is returned to a known state following a failure, reducing the requirement for manual intervention during restart.

By using the atomicity property, we can write a transaction program that emulates an atomic business transaction, such as a bank account withdrawal, a flight reservation, or a sale of stock shares. Each of these business actions requires updating multiple data items. By implementing the business action by a transaction, we ensure that either all the updates are performed or none are.

The successful completion of a transaction is called **commit**. The failure of a transaction is called **abort**.

Handling Real-World Operations

During its execution, a transaction may produce output that is displayed back to the user. However, since the transaction program is all-or-nothing, until the transaction actually commits, any results that the transaction might display to the user should not be taken seriously, because it's still possible that the transaction will abort. Anything displayed on the display device could be wiped out in the database on abort.

Thus, any value that the transaction displays may be used by the end-user only if the transaction commits and not if the transaction aborts. This requires some care on the part of users (see Figure 1.4). If the system actually displays some of the results of a transaction before the transaction commits, and if the user utilizes any of these results as input to another transaction, then we have a problem. If the first transaction aborts and the second transaction commits, then the all-or-nothing property has been broken. That is, some of the results of the first transaction will be reflected in the results of the second transaction. But other results of the first transaction, such as its database updates, were not performed because the transaction aborted.

Some systems solve this problem simply by not displaying the result of a transaction until after the transaction commits, so the user can't inadvertently make use of the transaction's output and then have it subsequently abort. But this too has its problems (see Figure 1.5): If the transaction commits before displaying any of its results, and the system crashes before the transaction actually displays any of the results, then the user won't get a chance to see the output. Again, the transaction is not all-or-nothing; it executed all its database updates before it committed, but did not display its output.

We can make the problem more concrete by looking at it in the context of an automated teller machine (ATM) (see Figure 1.6). The output, for example, may be an operation that dispenses $100 from the ATM. If the system dispenses the $100 before the transaction commits, and the transaction ends up aborting, then the

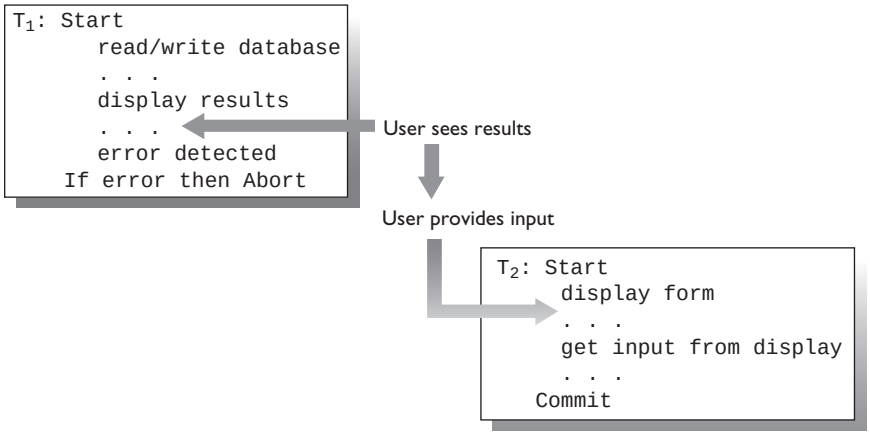

FIGURE 1.4

Reading Uncommitted Results. The user read the uncommitted results of transaction T_1 and fed them as input to transaction T_2. Since T_1 aborts, the input to T_2 is incorrect.

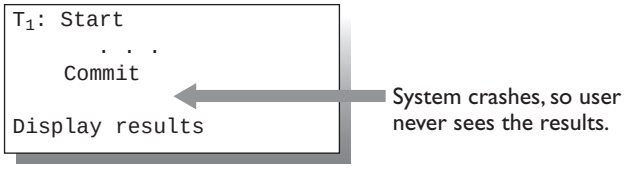

FIGURE 1.5

Displaying Results after Commits. This solves the problem of Figure 1.4, but if the transaction crashes before displaying the results, the results are lost forever.

bank gives up the money but does not record that fact in the database. If the transaction commits and the system fails before it dispenses the $100, then the database says the $100 was given to the customer, but in fact the customer never got the money. In both cases, the transaction's behavior is not all-or-nothing.

A closely-related problem is that of ensuring that each transaction executes exactly once. To do this, the transaction needs to send an acknowledgment to its caller, such as sending a message to the ATM to dispense money, if and only if it commits. However, sending this acknowledgment is not enough to guarantee exactly-once behavior because the caller cannot be sure how to interpret the absence of an acknowledgment. If the caller fails to receive an acknowledgment, it might be because the transaction aborted, in which case the caller needs to resubmit a request to run a transaction (to ensure the transaction executes once). Or it might be that the transaction committed but the acknowledgment got lost, in which case the caller must not resubmit a request to run the transaction because that would cause the transaction to execute twice. So if the caller wants exactly-once behavior, it needs to be sure that a transaction did not and will not commit before it's safe to resubmit the request to run the transaction.

Although these seem like unsolvable problems, they can actually be solved using persistent queues, which we'll describe in some detail in Chapter 4.

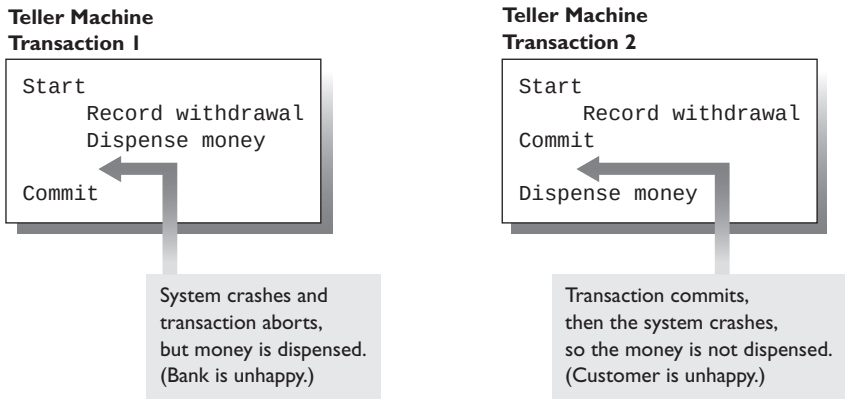

FIGURE 1.6

The Problem of Getting All-or-Nothing Behavior with Real-World Operations. Whether the program dispenses money before or after it commits, it's possible that only one of the operations executes: dispense the money or record the withdrawal.

Compensating Transactions

Commitment is an irrevocable action. Once a transaction is committed, it can no longer be aborted. People do make mistakes, of course. So it may turn out later that it was a mistake to have executed a transaction that committed. At this point, the only course of action is to run another transaction that reverses the effect of the one that committed. This is called a **compensating transaction**. For example, if a deposit transaction was in error, then one can later run a withdrawal transaction that reverses its effect.

Sometimes, a perfect compensation is impossible, because the transaction performed some irreversible act. For example, it may have caused a paint gun to spray-paint a part the wrong color, and the part is long gone from the paint gun's work area when the error is detected. In this case, the compensating transaction may be to record the error in a database and send an e-mail message to someone who can take appropriate action.

Virtually any transaction can be executed incorrectly. So a well-designed TP application should include a compensating transaction type for every type of transaction.

Multistep Business Processes

Some business activities do not execute as a single transaction. For example, the activity of recording an order typically executes in a separate transaction from the one that processes the order. Since recording an order is relatively simple, the system can give excellent response time to the person who entered the order. The processing of the order usually requires several time-consuming activities that may require multiple transactions, such as checking the customer's credit, forwarding the order to a warehouse that has the requested goods in stock, and fulfilling the order by picking, packing, and shipping it.

Even though the business process executes as multiple transactions, the user may still want atomicity. Since multiple transactions are involved, this often requires compensating transactions. For example, if an order is accepted by the system in one transaction, but later on another transaction determines that the order can't be fulfilled, then a compensating transaction is needed to reverse the effect of the transaction that accepted the order. To avoid an unhappy customer, this often involves the universal compensating transaction, namely, an apology and a free gift certificate. It might also involve offering the customer a choice of either cancelling or telling the retailer to hold the order until the requested items have been restocked.

Transactional middleware can help manage the execution of multistep business processes. For example, it can keep track of the state of a multistep process, so if the process is unable to complete then the middleware can invoke compensating transactions for the steps that have already executed. These functions and others are discussed in Chapter 5, *Business Process Management*.

Consistency

A second property of transactions is consistency—a transaction program should maintain the consistency of the database. That is, if you execute the transaction all by itself on a database that's initially consistent, then when the transaction finishes executing the database is again consistent.

By consistent, we mean "internally consistent." In database terms, this means that the database at least satisfies all its integrity constraints. There are several kinds of integrity constraints that database systems can typically maintain:

- All primary key values are unique (e.g., no two employee records have the same employee number).
- The database has referential integrity, meaning that records reference only objects that exist (e.g., the Part record and Customer record that are referenced by an Order record really exist).
- Certain data values are in a particular range (e.g., age is less than 120 and social security number is not null).

There are other kinds of integrity constraints that database systems typically cannot maintain but may nevertheless be important, such as the following:

- The sum of expenses in each department is less than or equal to the department's budget.
- The salary of an employee is bounded by the salary range of the employee's job level.
- The salary of an employee cannot decrease unless the employee is demoted to a lower job level.

Ensuring that transactions maintain the consistency of the database is good programming practice. However, unlike atomicity, isolation, and durability, consistency is a responsibility shared between transaction programs and the TP system that executes those programs. That is, a TP system ensures that transactions are atomic, isolated, and durable, whether or not they are programmed to preserve consistency. Thus, strictly speaking, the ACID test for transaction systems is a bit too strong, because the TP system does its part for the C in ACID only by guaranteeing AID. It's the application programmer's responsibility to ensure the transaction program preserves consistency.

There are consistency issues that reach out past the TP system and into the physical world that the TP application describes. An example is the constraint that the number of physical items in inventory equals the number of items on the warehouse shelf. This constraint depends on actions in the physical world, such as correctly reporting the restocking and shipment of items in the warehouse. Ultimately, this is what the enterprise regards as consistency.

Isolation

The third property of a transaction is called **isolation**. We say that a set of transactions is isolated if the effect of the system running them is the same as if the system ran them one at a time. The technical definition of isolation is serializability. An execution is **serializable** (meaning isolated) if its effect is the same as running the transactions serially, one after the next, in sequence, with no overlap in executing any two of them. This has the same effect as running the transactions one at a time.

A classic example of a non-isolated execution is a banking system, where two transactions each try to withdraw the last $100 in an account. If both transactions read the account balance before either of them updates it,

then both transactions will determine there's enough money to satisfy their requests, and both will withdraw the last $100. Clearly, this is the wrong result. Moreover, it isn't a serializable result. In a serial execution, only the first transaction to execute would be able to withdraw the last $100. The second one would find an empty account.

Notice that isolation is different from atomicity. In the example, both transactions executed completely, so they were atomic. However, they were not isolated and therefore produced undesirable behavior.

If the execution is serializable, then from the point of view of an end-user who submits a request to run a transaction, the system looks like a standalone system that's running that transaction all by itself. Between the time he or she runs two transactions, other transactions from other users may run. But during the period that the system is processing that one user's transaction, the user has the illusion that the system is doing no other work. This is only an illusion. It's too inefficient for the system to actually run transactions serially, because there is a lot of internal parallelism in the system that must be exploited by running transactions concurrently.

If each transaction preserves consistency, then any serial execution (i.e., sequence) of such transactions preserves consistency. Since each serializable execution is equivalent to a serial execution, a serializable execution of the transactions will preserve database consistency too. It is the combination of transaction consistency and isolation that ensures that the execution of a set of transactions preserves database consistency.

The database typically sets locks on data accessed by each transaction. The effect of setting the locks is to make the execution appear to be serial. In fact, internally, the system is running transactions in parallel, but through this locking mechanism the system gives the illusion that the transactions are running serially, one after the next. In Chapter 6, we will describe those mechanisms in more detail and present the rather subtle argument why locking actually produces serializable executions.

A common misconception is that serializability isn't important because the database system will maintain consistency by enforcing integrity constraints. However, as we saw in the previous section on consistency, there are many consistency constraints that database systems can't enforce. Moreover, sometimes users don't tell the database system to enforce certain constraints because they degrade performance. The last line of defense is that the transaction program itself maintains consistency and that the system guarantees serializability.

Durability

The fourth property of a transaction is durability. **Durability** means that when a transaction completes executing, all its updates are stored in **stable storage**; that is, storage that will survive the failure of power or the operating system. Today, stable storage (also called **nonvolatile** or **persistent storage**) typically consists of magnetic disk drives, though solid-state disks that use flash memory are making inroads as a viable alternative. Even if the transaction program fails, or the operating system fails, once the transaction has committed, its results are durably stored on stable storage and can be found there after the system recovers from the failure.

Durability is important because each transaction usually is providing a service to the user that amounts to a contract between the user and the enterprise that is providing the service. For example, if you're moving money from one account to another, once you get a reply from the transaction saying that it executed, you expect that the result is permanent. It's a legal agreement between the user and the system that the money has been moved between these two accounts. So it's essential that the transaction actually makes sure that the updates are stored on some stable storage device, to ensure that the updates cannot possibly be lost after the transaction finishes executing. Moreover, the durability of the result must be maintained for a long period, until it is explicitly overwritten or deleted by a later transaction. For example, even if a checking account is unused for several years, the owner expects to find her money there the next time she accesses it.

The durability property usually is obtained by having the TP system append a copy of all the transaction's updates to a log file while the transaction program is running. When the transaction program issues the commit operation, the system first ensures that all the records written to the log file are out on stable storage, and then

returns to the transaction program, indicating that the transaction has indeed committed and that the results are durable. The updates may be written to the database right away, or they may be written a little later. However, if the system fails after the transaction commits and before the updates go to the database, then after the system recovers from the failure it must repair the database. To do this, it reads the log and checks that each update by a committed transaction actually made it to the database. If not, it reapplies the update to the database. When this recovery activity is complete, the system resumes normal operation. Thus, after the system recovers, any new transaction will read a database state that includes all the updates of transactions that committed before the failure (as well as those that committed after the recovery). We describe log-based recovery algorithms in Chapter 7.

1.4 TWO-PHASE COMMIT

When a transaction updates data on two or more database systems, we still have to ensure the atomicity property, namely, that either both database systems durably install the updates or neither does. This is challenging, because the database systems can independently fail and recover. This is certainly a problem when the database systems reside on different nodes of a distributed system. But it can even be a problem on a single machine if the database systems run as server processes with private storage since the processes can fail independently. The solution is a protocol called **two-phase commit** (**2PC**), which is executed by a module called the **transaction manager**.

The crux of the problem is that a transaction can commit its updates on one database system, but a second database system can fail before the transaction commits there too. In this case, when the failed system recovers, it must be able to commit the transaction. To commit the transaction, the recovering system must have a copy of the transaction's updates that executed there. Since a system can lose the contents of main memory when it fails, it must store a durable copy of the transaction's updates before it fails, so it will have them after it recovers. This line of reasoning leads to the essence of two-phase commit: Each database system accessed by a transaction must durably store its portion of the transaction's updates before the transaction commits anywhere. That way, if a system S fails after the transaction commits at another system S' but before the transaction commits at S, then the transaction can commit at S after S recovers (see Figure 1.7).

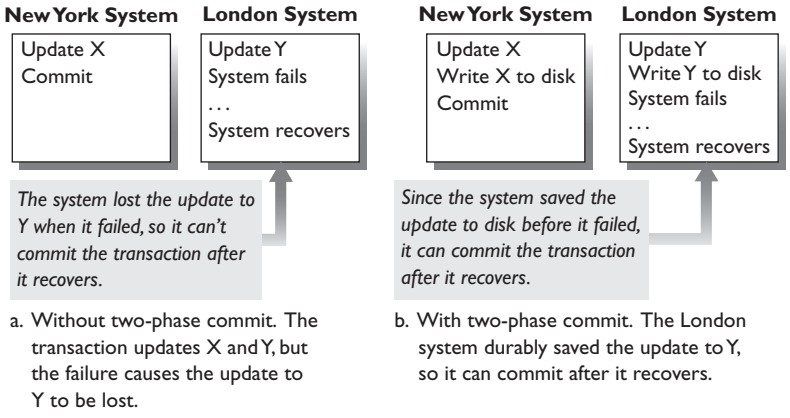

FIGURE 1.7

How Two-Phase Commit Ensures Atomicity. With two-phase commit, each system durably stores its updates before the transaction commits, so it can commit the transaction when it recovers.

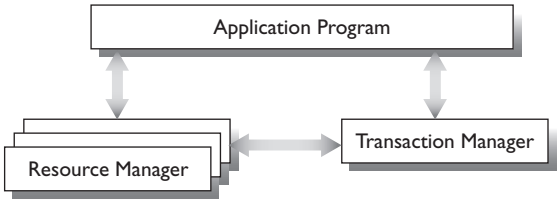

FIGURE 1.8

X/Open Transaction Model (XA). The transaction manager processes Start, Commit, and Abort. It talks to resource managers to run two-phase commit.

To understand two-phase commit, it helps to visualize the overall architecture in which the transaction manager operates. The standard model, shown in Figure 1.8, was introduced by IBM's CICS and popularized by Oracle's Tuxedo and X/Open (now part of The Open Group, see Chapter 10). In this model, the transaction manager talks to applications, resource managers, and other transaction managers. The concept of "resource" includes databases, queues, files, messages, and other shared objects that can be accessed within a transaction. Each resource manager offers operations that must execute only if the transaction that called the operations commits.

The transaction manager processes the basic transaction operations for applications: Start, Commit, and Abort. An application calls Start to begin executing a new transaction. It calls Commit to ask the transaction manager to commit the transaction. It calls Abort to tell the transaction manager to abort the transaction.

The transaction manager is primarily a bookkeeper that keeps track of transactions in order to ensure atomicity when more than one resource is involved. Typically, there's one transaction manager on each node of a distributed computer system. When an application issues a Start operation, the transaction manager dispenses a unique ID for the transaction called a **transaction identifier**. During the execution of the transaction, it keeps track of all the resource managers that the transaction accesses. This requires some cooperation with the application, resource managers, and communication system. Whenever the transaction accesses a new resource manager, somebody has to tell the transaction manager. This is important because when it comes time to commit the transaction, the transaction manager has to know all the resource managers to talk to in order to execute the two-phase commit protocol.

When a transaction program finishes execution and issues the commit operation, that commit operation goes to the transaction manager, which processes the operation by executing the two-phase commit protocol. Similarly, if the transaction manager receives an abort operation, it tells the resource managers to undo all the transaction's updates; that is, to abort the transaction at each resource manager. Thus, each resource manager must understand the concept of transaction, in the sense that it undoes or permanently installs the transaction's updates depending on whether the transaction aborts or commits.

When running two-phase commit, the transaction manager sends out two rounds of messages—one for each phase of the commitment activity. In the first round of messages it tells all the resource managers to prepare to commit by writing a copy of the results of the transaction to stable storage, but not actually to commit the transaction. At this point, the resource managers are said to be **prepared to commit**. When the transaction manager gets acknowledgments back from all the resource managers, it knows that the whole transaction has been prepared. That is, it knows that all resource managers stored a durable copy of the transaction's updates but none of them have committed the transaction. So it sends a second round of messages to tell the resource managers to actually commit. Figure 1.9 gives an example execution of two-phase commit with two resource managers involved.

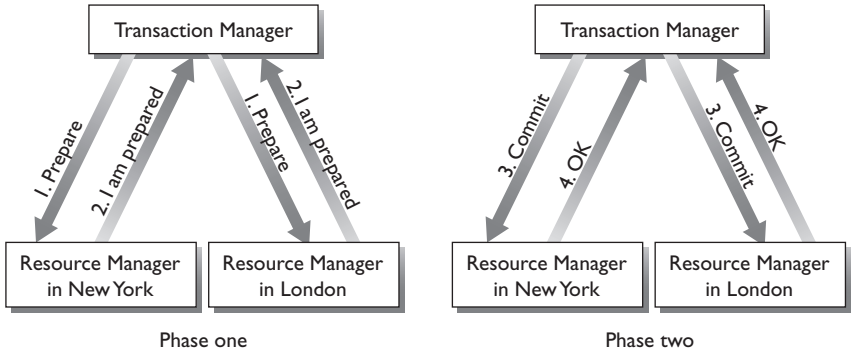

FIGURE 1.9

The Two-Phase Commit Protocol. In Phase One, every resource manager durably saves the transaction's updates before replying "I am Prepared." Thus, all resource managers have durably stored the transaction's updates before any of them commits in phase two.

Two-phase commit avoids the problem in Figure 1.7(a) because all resource managers have a durable copy of the transaction's updates before any of them commit. Therefore, even if a system fails during the commitment activity, as the London system did in the figure, it can commit the transaction after it recovers. However, to make this all work, the protocol must handle every possible failure and recovery scenario. For example, in Figure 1.7(b), it must tell the London system to commit the transaction. The details of how two-phase commit handles all these scenarios is described in Chapter 8.

Two-phase commit is required whenever a transaction accesses two or more resource managers. Thus, one key question that designers of TP applications must answer is whether or not to distribute their transaction programs among multiple resources. Using two-phase commit adds overhead (due to two-phase commit messages), but the option to distribute can provide better scalability (adding more systems to increase capacity) and availability (since one system can fail while others remain operational).

1.5 TRANSACTION PROCESSING PERFORMANCE

Performance is a critical aspect of TP systems. No one likes waiting more than a few seconds for an automated teller machine to dispense cash or for a hotel web site to accept a reservation request. So response time to end-users is one important measure of TP system performance. Companies that rely on TP systems, such as banks, airlines, and commercial web sites, also want to get the most transaction throughput for the money they invest in a TP system. They also care about system scalability; that is, how much they can grow their system as their business grows.

It's very challenging to configure a TP system to meet response time and throughput requirements at minimum cost. It requires choosing the number of systems, how much storage capacity they'll have, which processing and database functions are assigned to each system, and how the systems are connected to displays and to each other. Even if you know the performance of the component products being assembled, it's hard to predict how the overall system will perform. Therefore, users and vendors implement benchmarks to obtain guidance on how to configure systems and to compare competing products.

Vendor benchmarks are defined by an independent consortium called the Transaction Processing Performance Council (TPC; www.tpc.org). The benchmarks enable apples-to-apples comparisons of different vendors' hardware

and software products. Each TPC benchmark defines standard transaction programs and characterizes a system's performance by the throughput that the system can process under certain workload conditions, database size, response time guarantees, and so on. Published results must be accompanied by a **full disclosure report**, which allows other vendors to review benchmark compliance and gives users more detailed performance information beyond the summary performance measures.

The benchmarks use two main measures of a system's performance, throughput, and cost-per-throughput-unit. Throughput is the maximum throughput it can attain, measured in **transactions per second (tps)** or **transactions per minute (tpm)**. Each benchmark defines a response time requirement for each transaction type (typically 1–5 seconds). The throughput can be measured only when 90% of the transactions meet their response time requirements and when the average of all transaction response times is less than their response time requirement. The latter ensures that all transactions execute within an acceptable period of time.

As an aside, Internet web sites usually measure 90% and 99% response times. Even if the average performance is fast, it's bad if one in a hundred transactions is too slow. Since customers often run multiple transactions, that translates into several percent of customers receiving poor service. Many such customers don't return.

The benchmarks' cost-per-throughput-unit is measured in dollars per tps or tpm. The cost is calculated as the list purchase price of the hardware and software, plus three years' vendor-supplied maintenance on that hardware and software (called the **cost of ownership**).

The definitions of TPC benchmarks are worth understanding to enable one to interpret TPC performance reports. Each of these reports, published on the TPC web site, is the result of a system benchmark evaluation performed by a system vendor and subsequently validated by an independent auditor. Although their main purpose is to allow customers to compare TP system products, these reports are also worth browsing for educational reasons, to give one a feel for the performance range of state-of-the-art systems. They are also useful as guidance for the design and presentation of a custom benchmark study for a particular user application.

The TPC-A and TPC-B Benchmarks

The first two benchmarks promoted by TPC, called TPC-A and TPC-B, model an ATM application that debits or credits a checking account. When TPC-A/B were introduced, around 1989, they were carefully crafted to exercise the main bottlenecks customers were experiencing in TP systems. The benchmark was so successful in encouraging vendors to eliminate these bottlenecks that within a few years nearly all database systems performed very well on TPC-A/B. Therefore, the benchmarks were retired and replaced by TPC-C in 1995. Still, it's instructive to look at the bottlenecks the benchmarks were designed to exercise, since these bottlenecks can still arise today on a poorly designed system or application.

Both benchmarks run the same transaction program. The only difference is that TPC-A includes terminals and a network in the overall system, while TPC-B does not. In both cases, the transaction program performs the sequence of operations shown in Figure 1.10 (except that TPC-B does not perform the read/write terminal operations).

In TPC-A/B, the database consists of:

- Account records, one record for each customer's account (total of 100,000 accounts)
- A teller record for each teller, which stores the amount of money in the teller's cash drawer (total of 10 tellers)
- One record for each bank branch (one branch minimum), which contains the sum of all the accounts at that branch
- A history file, which records a description of each transaction that actually executes

```
Start
  Read message from terminal (100 bytes)
  Read and write account record (random access)
  Write history record (sequential access)
  Read and write teller record (random access)
  Read and write branch record (random access)
  Write message to terminal (200 bytes)
Commit
```

FIGURE 1.10

TPC-A/B Transaction Program. The program models a debit/credit transaction for a bank.

The transaction reads a 100-byte input message, including the account number and amount of money to withdraw or deposit. The transaction uses that input to find the account record and update it appropriately. It updates the history file to indicate that this transaction has executed. It updates the teller and bank branch records to indicate the amount of money deposited or withdrawn at that teller and bank branch, respectively. Finally, for TPC-A, it sends a message back to the display device to confirm the completion of the transaction.

The benchmark exercises several potential bottlenecks on a TP system:

- There's a large number of account records. The system must have 100,000 account records for each transaction per second it can perform. To randomly access so many records, the database must be indexed.
- The end of the history file can be a bottleneck, because every transaction has to write to it and therefore to lock and synchronize against it. This synchronization can delay transactions.
- Similarly, the branch record can be a bottleneck, because all of the tellers at each branch are reading and writing it. However, TPC-A/B minimizes this effect by requiring a teller to execute a transaction only every 10 seconds.

Given a fixed configuration, the performance and price/performance of any TP application depends on the amount of computer resources needed to execute it: the number of processor instructions, I/Os to stable storage, and communications messages. Thus, an important step in understanding the performance of any TP application is to count the resources required for each transaction. In TPC-A/B, for each transaction a high performance implementation uses a few hundred thousand instructions, two or three I/Os to stable storage, and two interactions with the display. When running these benchmarks, a typical system spends more than half of the processor instructions inside the database system and maybe another third of the instructions in message communications between the parts of the application. Only a small fraction of the processor directly executes the transaction program. This isn't very surprising, because the transaction program mostly just sends messages and initiates database operations. The transaction program itself does very little, which is typical of many TP applications.

The TPC-C Benchmark

The TPC-C benchmark was introduced in 1992. It is based on an order-entry application for a wholesale supplier. Compared to TPC-A/B, it includes a wider variety of transactions, some "heavy weight" transactions (which do a lot of work), and a more complex database.

The database centers around a *warehouse*, which tracks the *stock* of *items* that it *supplies* to *customers* within a sales *district*, and tracks those customers' *orders*, which consist of *order-lines*. The database size is proportional to the number of warehouses (see Table 1.1).

Table 1.1 Database for the TPC-C Benchmark. The database consists of the tables in the left column, which support an order-entry application

Table Name	Number of Rows per Warehouse	Bytes-per-Row	Size of Table (in bytes) per Warehouse
Warehouse	1	89	.089 K
District	10	95	.95 K
Customer	30 K	655	19.65 K
History	30 K	46	1.38 K
Order	30 K	24	720 K
New-Order	9 K	8	72 K
Order-Line	300 K	54	16.2 M
Stock	100 K	306	306 M
Item	100 K	82	8.2 M

There are five types of transactions:

- **New-Order:** To enter a new order, first retrieve the records describing the given warehouse, customer, and district, and then update the district (increment the next available order number). Insert a record in the Order and New-Order tables. For each of the 5 to 15 (average 10) items ordered, retrieve the item record (abort if it doesn't exist), retrieve and update the stock record, and insert an order-line record.
- **Payment:** To enter a payment, first retrieve and update the records describing the given warehouse, district, and customer, and then insert a history record. If the customer is identified by name, rather than id number, then additional customer records (average of two) must be retrieved to find the right customer.
- **Order-Status:** To determine the status of a given customer's latest order, retrieve the given customer record (or records, if identified by name, as in Payment), and retrieve the customer's latest order and corresponding order-lines.
- **Delivery:** To process a new order for each of a warehouse's 10 districts, get the oldest new-order record in each district, delete it, retrieve and update the corresponding customer record, order record, and the order's corresponding order-line records. This can be done as one transaction or 10 transactions.
- **Stock-Level:** To determine, in a warehouse's district, the number of recently sold items whose stock level is below a given threshold, retrieve the record describing the given district (which has the next order number). Retrieve order lines for the previous 20 orders in that district, and for each item ordered, determine if the given threshold exceeds the amount in stock.

The transaction rate metric is the number of New-Order transactions per minute, denoted **tpmC**, given that all the other constraints are met. The New-Order, Payment, and Order-Status transactions have a response time requirement of five seconds. The Stock-Level transaction has a response time of 20 seconds and has relaxed consistency requirements. The Delivery transaction runs as a periodic batch. The workload requires executing an equal number of New-Order and Payment transactions, and one Order-Status, Delivery, and Stock-Level transaction for every 10 New-Orders.

Table 1.2 TPC-E Transaction Types

Transaction Type	Percent of Transactions	Database Tables Accessed	Description
Trade Order	10.1%	17	Buy or sell a security
Trade Result	10%	15	Complete the execution of a buy or sell order
Trade Status	19%	6	Get the status of an order
Trade Update	2%	6	Make corrections to a set of trades
Customer Position	13%	7	Get the value of a customer's assets
Market Feed	1%	2	Process an update of current market activity (e.g., ticker tape)
Market Watch	18%	4	Track market trends (e.g., for a customer's "watch list")
Security Detail	14%	12	Get a detailed data about a security
Trade Lookup	8%	6	Get information about a set of trades
Broker Volume	4.9%	6	Get a summary of the volume and value of pending orders of a set of brokers

The TPC-C workload is many times heavier per transaction than TPC-A/B and exhibits higher contention for shared data. Moreover, it exercises a wider variety of performance-sensitive functions, such as deferred transaction execution, access via secondary keys, and transaction aborts. It is regarded as a more realistic workload than TPC-A/B, which is why it replaced TPC-A/B as the standard TP systems benchmark.

The TPC-E Benchmark

The TPC-E benchmark was introduced in 2007. Compared to TPC-C, it represents larger and more complex databases and transaction workloads that are more representative of current TP applications. And it uses a storage configuration that is less expensive to test and run. It is based on a stock trading application for a brokerage firm where transactions are related to stock trades, customer inquiries, activity feeds from markets, and market analysis by brokers. Unlike previous benchmarks, TPC-E does not include transactional middleware components and solely measures database performance.

TPC-E includes 10 transaction types, summarized in Table 1.2, which are a mix of read-only and read-write transactions. For each type, the table shows the percentage of transactions of that type and the number of database tables it accesses, which give a feeling for the execution cost of the type.

There are various parameters that introduce variation into the workload. For example, trade requests are split 50-50 between buy and sell and 60-40 between market order and limit order. In addition, customers are assigned to one of three tiers, depending on how often they trade securities—the higher the tier, the more accounts per customer and trades per customer.

The database schema has 33 tables divided into four sets: market data (11 tables), customer data (9 tables), broker data (9 tables), and static reference data (4 tables). Most tables have fewer than six columns and less than 100 bytes per row. At the extremes, the Customer table has 23 columns, and several tables store text information with hundreds of bytes per row (or even more for the News Item table).

A driver program generates the transactions and their inputs, submits them to a test system, and measures the rate of completed transactions. The result is the **measured** transactions per second (tpsE), which is the number of Trade Result transactions executed per second, given the mix of the other transaction types. Each transaction type has a response time limit of one to three seconds, depending on transaction type. In contrast to TPC-C, application functions related to front-end programs are excluded. Thus, the results measure the server-side database management system. Like previous TPC benchmarks, TPC-E includes a measure for the cost per transaction per second ($/tpsE).

TPC-E provides data generation code to initialize the database with the result of 300 days of initial trading, daily market closing price information for five years, and quarterly company report data for five years. Beyond that, the database size scales up as a function of the **nominal** tpsE, which is the transaction rate the benchmark sponsor is aiming for. The measured tpsE must be within 80 to 102% of the nominal tpsE. The database must have 500 customers for each nominal tpsE. Other database tables scale relative to the number of customer rows. For example, for each 1000 Customers, there must be 685 Securities and 500 Companies. Some tables include a row describing each trade and therefore grow quite large for a given run.

Compared to TPC-C, TPC-E is a more complex workload. It makes heavier use of SQL database features, such as referential integrity and transaction isolation levels (to be discussed in Chapter 6). It uses a more complex SQL schema. Transactions execute more complex SQL statements and several of them have to make multiple calls to the database, which cannot be batched in one round-trip. And there is no trivial partitioning of the database that will enable scalability (to be discussed in Section 2.6). Despite all this newly introduced complexity, the benchmark generates a much lower I/O load than TPC-C for a comparable transaction rate. This makes the benchmark cheaper to run, which is important to vendors when they run high-end scalability tests where large machine configurations are needed.

In addition to its TP benchmarks, the TPC publishes a widely used benchmark for decision support systems, TPC-H. It also periodically considers new TP benchmark proposals. Consult the TPC web site, www.tpc.org, for current details.

1.6 AVAILABILITY

Availability is the fraction of time a TP system is up and running and able to do useful work—that is, it isn't down due to hardware or software failures, operator errors, preventative maintenance, power failures, or the like. Availability is an important measure of the capability of a TP system because the TP application usually is offering a service that's "mission critical," one that's essential to the operation of the enterprise, such as airline reservations, managing checking accounts in a bank, processing stock transactions in a stock exchange, or offering a retail storefront on the Internet. Obviously, if this type of system is unavailable, the business stops operating. Therefore, the system *must* operate nearly all the time.

Just how highly available does a system have to be? We see from the table in Figure 1.11 that if the system is available 96% of the time, that means it's down nearly an hour a day. That's too much time for many types of businesses, which would consider 96% availability to be unacceptable.

An availability of 99% means that the system is down about 100 minutes per week (i.e., 7 days/week $\times$ 24 hours/day $\times$ 60 minutes/hour $\times$ 1/100). Many TP applications would find this unacceptable if it came in one 100-minute period of unavailability. It might be tolerable, provided that it comes in short outages of just a few minutes at a time. But in many cases, even this may not be tolerable, for example in the operation of a stock exchange where short periods of downtime can produce big financial losses.

An availability of 99.9% means that the system is down for about an hour per month, or under two minutes per day. Further, 99.999% availability means that the system is down five minutes a year. That number

Downtime	Availability (%)
1 hour/day	95.8
1 hour/week	99.41
1 hour/month	99.86
1 hour/year	99.9886
1 hour/20 years	99.99942

FIGURE 1.11

Downtime at Different Availability Level. The number of nines after the decimal point is of practical significance.

may seem incredibly ambitious, but it *is* attainable; telephone systems typically have that level of availability. People sometimes talk about availability in terms of the number of 9s that are attained; for example, "five 9s" means 99.999% available.

Some systems need to operate for only part of the day, such as 9 AM to 5 PM on weekdays. In that case, availability usually is measured relative to the hours when the system is expected to be operational. Thus, 99.9% availability means that it is down at most 2.4 minutes per week (i.e., 40 hours/week $\times$ 60 minutes/hour $\times$ 1/1000).

Today's TP system customers typically expect availability levels of at least 99%, although it certainly depends on how much money they're willing to spend. Generally, attaining high availability requires attention to four factors:

- The environment—making the physical environment more robust to avoid failures of power, communications, air conditioning, and the like
- System management—avoiding failures due to operational errors by system managers and vendors' field service
- Hardware—having redundant hardware, so that if some component fails, the system can immediately and automatically replace it with another component that's ready to take over
- Software—improving the reliability of software and ensuring it can automatically and quickly recover after a failure

This book is about software, and regrettably, of the four factors, software is the major contributor to availability problems. Software failures can be divided into three categories: application failures, database system failures, and operating system failures.

Because we're using transactions, when an application fails, any uncommitted transaction it was executing aborts automatically. Its updates are backed out, because of the atomicity property. There's really nothing that the system has to do other than re-execute the transaction after the application is running again.

When the database system fails, all the uncommitted transactions that were accessing the database system at that time have to abort, because their updates may be lost during the database system failure. A system management component of the operating system, database system, or transactional middleware has to detect the failure of the database system and tell the database system to reinitialize itself. During the reinitialization process, the database system backs out the updates of all the transactions that were active at the time of the failure, thereby getting the database into a clean state, where it contains the results only of committed transactions.

A failure of the operating system requires it to reboot. All programs, applications, and database systems executing at the time of failure are now dead. Everything has to be reinitialized after the operating system reboots. On an ordinary computer system all this normally takes between several minutes and an hour, depending on how big the system is, how many transactions were active at the time of failure, how long it takes to back out the uncommitted transactions, how efficient the initialization program is, and so on. Very high availability systems, such as those intended to be available in excess of 99%, typically are designed for very fast recovery. Even when

they fail, they are down only for a very short time. They usually use some form of replicated processing to get this fast recovery. When one component fails, they quickly delegate processing work to a copy of the component that is ready and waiting to pick up the load.

The transaction abstraction helps the programmer quite a bit in attaining high availability, because the system is able to recover into a clean state by aborting transactions. And it can continue from where it left off by rerunning transactions that aborted as a result of the failure. Without the transaction abstraction, the recovery program would have to be application-specific. It would have to analyze the state of the database at the time of the failure to figure out what work to undo and what to rerun. We discuss high availability issues and techniques in more detail in Chapter 7, and replication technology in Chapter 9.

In addition to application, database system, and operating system failures, operator errors are a major contributor to unplanned downtime. Many of these errors can be attributed to system management software that is hard to understand and use. If the software is difficult to tune, upgrade, or operate, then operators make mistakes. The ideal system management software is fully automated and requires no human intervention for such routine activities.

1.7 STYLES OF SYSTEMS

We've been talking about TP as a style of *application*, one that runs short transaction programs that access a shared database. TP is also a style of *system*, a way of configuring software components to do the type of work required by a TP application. It's useful to compare this style of system with other styles that you may be familiar with, to see where the differences are and why TP systems are constructed differently from the others. There are several other kinds of systems that we can look at here:

- Batch processing systems, where you submit a job and later receive output in the form of a file
- Real-time systems, where you submit requests to do a small amount of work that has to be done before some very early deadline
- Data warehouse systems, where reporting programs and *ad hoc* queries access data that is integrated from multiple data sources

Designing a system to perform one of these types of processing is called *system engineering*. Rather than engineering a specific component, such as an operating system or a database system, you engineer an integrated system by combining different kinds of components to perform a certain type of work. Often, systems are engineered to handle multiple styles, but for the purposes of comparing and contrasting the different styles, we'll discuss them as if each type of system were running in a separately engineered environment. Let's look at requirements for each of these styles of computing and see how they compare to a TP system.

Batch Processing Systems

A batch is a set of requests that are processed together, often long after the requests were submitted. Data processing systems of the 1960s and early 1970s were primarily batch processing systems. Today, batch workloads are still with us. But instead of running them on systems dedicated for batch processing, they often execute on systems that also run a TP workload. TP systems can execute the batches during nonpeak periods, since the batch workload has flexible response-time requirements. To make the comparison between TP and batch clear, we will compare a TP system running a pure TP workload against a classical batch system running a pure batch workload, even though mixtures of the two are now commonplace.

A batch processing system executes each batch as a sequence of transactions, one transaction at a time. Since transactions execute serially there's no problem with serializability. By contrast, in a TP system many transactions can execute at the same time, and so the system has extra work to ensure serializability.

For example, computing the value of a stock market portfolio could be done as a batch application, running once a day after the close of financial markets. Computing a monthly bill for telephone customers could be a batch application, running daily for a different subset of the customer base each day. Generating tax reporting documents could be a batch application executed once per quarter or once per year.

The main performance measure of batch processing is throughput, that is, the amount of work done per unit of time. Response time is less important. A batch could take minutes, hours, or even days to execute. By contrast, TP systems have important response time requirements, because generally there's a user waiting at a display for the transaction's output.

A classical batch processing application takes its input as a record-oriented file whose records represent a sequence of request messages. Its output is also normally stored in a file. By contrast, TP systems typically have large networks of display devices for capturing requests and displaying results.

Batch processing can be optimized by ordering the input requests consistently with the order of the data in the database. For example, if the requests correspond to giving airline mileage credit for recent flights to mileage award customers, the records of customer flights can be ordered by mileage award account number. That way, it's easy and efficient to process the records by a merge procedure that reads the mileage award account database in account number order. By contrast, TP requests come in a random order. Because of the fast response time requirement, the system can't spend time sorting the input in an order consistent with the database. It has to be able to access the data randomly, in the order in which the data is requested.

Classical batch processing takes the request message file and existing database file(s) as input and produces a new master output database as a result of running transactions for the requests. If the batch processing program should fail, there's no harm done because the input file and input database are unmodified—simply throw out the output file and run the batch program again. By contrast, a TP system updates its database on-line as requests arrive. So a failure may leave the database in an inconsistent state, because it contains the results of uncompleted transactions. This atomicity problem for transactions in a TP environment doesn't exist in a batch environment.

Finally, in batch the load on the system is fixed and predictable, so the system can be engineered for that load. For example, you can schedule the system to run the batch at a given time and set aside sufficient capacity to do it, because you know exactly what the load is going to be. By contrast, a TP load generally varies during the day. There are peak periods when there's a lot of activity and slow periods when there's very little. The system has to be sized to handle the peak load and also designed to make use of extra capacity during slack periods.

Real-Time Systems

TP systems are similar to real-time systems, such as a system collecting input from a satellite or controlling a factory's shop floor equipment. TP essentially is a kind of real-time system, with a real-time response time demand of 1 to 2 seconds. It responds to a real-world process consisting of end-users interacting with display devices, which communicate with application programs accessing a shared database. So not surprisingly, there are many similarities between the two kinds of systems.

Real-time systems and TP systems both have predictable loads with periodic peaks. Real-time systems usually emphasize gathering input rather than processing it, whereas TP systems generally do both.

Due to the variety of real-world processes they control, real-time systems generally have to deal with more specialized devices than TP, such as laboratory equipment, factory shop floor equipment, or sensors and control systems in an automobile or airplane.

Real-time systems generally don't need or use special mechanisms for atomicity and durability. They simply process the input as quickly as they can. If they lose some of that input, they ignore the loss and keep on running. To see why, consider the example of a system that collects input from a monitoring satellite. It's not good if the system misses some of the data coming in. But the system certainly can't stop operating to go back to fix things up like a TP system would do—the data keeps coming in and the system must do its best to continue processing it. By contrast, a TP environment can generally stop accepting input for a short time or can buffer the input for awhile. If there is a failure, it can stop collecting input, run a recovery procedure, and then resume processing input. Thus, the fault-tolerance requirements between the two types of systems are rather different.

Real-time systems are generally not concerned with serializability. In most real-time applications, processing of input messages involves no access to shared data. Since the processing of two different inputs does not affect each other, even if they're processed concurrently, they'll behave like a serial execution. No special mechanisms, such as locking, are needed. When processing real-time inputs to shared data, the notion of serializability is as relevant as it is to TP. However, in this case, real-time applications generally make direct use of low-level synchronization primitives for mutual exclusion, rather than relying on a general-purpose synchronization mechanism that is hidden behind the transaction abstraction.

Data Warehouse Systems

TP systems process the data in its raw state as it arrives. **Data warehouse systems** integrate data from multiple sources into a database suitable for querying.

For example, a distribution company decides each year how to allocate its marketing and advertising budget. It uses a TP system to process sales orders that includes the type and value of each order. The customer database tells each customer's location, annual revenue, and growth rate. The finance database includes cost and income information, and tells which product lines are most profitable. The company pulls data from these three data sources into a data warehouse. Business analysts can query the data warehouse to determine how best to allocate promotional resources.

Data warehouse systems execute two kinds of workloads: a batch workload to extract data from the sources, cleaning the data to reconcile discrepancies between them, transforming the data into a common shape that's convenient for querying, and loading it into the warehouse; and queries against the warehouse, which can range from short interactive requests to complex analyses that generate large reports. Both of these workloads are quite different than TP, which consists of short updates and queries. Also unlike TP, a data warehouse's content can be somewhat out-of-date, since users are looking for trends that are not much affected by the very latest updates. In fact, sometimes it's important to run on a static database copy, so that the results of successive queries are comparable. Running queries on a data warehouse rather than a TP database is also helpful for performance reasons, since data warehouse queries would slow down update transactions, a topic we'll discuss in some detail in Chapter 6. Our comparison of system styles so far is summarized in Figure 1.12.

Other System Types

Two other system types that are related to TP are timesharing and client-server.

Timesharing

In a timesharing system, a display device is connected to an operating system process, and within that process the user can invoke programs that interact frequently with the display. Before the widespread use of PCs, when timesharing systems were popular, TP systems often were confused with timesharing, because they both

	Transaction Processing	Batch	Real-time	Data Warehouse
Isolation	serializable, multi-programmed execution	serial, uni-programmed execution	no transaction concept	no transaction concept
Workload	high variance	predictable	predictability depends on the application	predictable loading, high variance queries
Performance metric	response time and throughput	throughput	response time, throughput, missed deadlines	throughput for loading, response time for queries
Input	network of display devices submitting requests	record-oriented file	network of devices submitting data and operations	network of display devices submitting queries
Data Access	random access	accesses sorted to be consistent with database order	unconstrained	possibly sorted for loading, unconstrained for queries
Recovery	after failure, ensure database has committed updates and no others	after failure, rerun the batch to produce a new master file	application's responsibility	application's responsibility

FIGURE 1.12

Comparison of System Types. Transaction processing has different characteristics than the other styles, and therefore requires systems that are specially engineered to the purpose.

involve managing lots of display devices connected to a common server. But they're really quite different in terms of load, performance requirements, and availability requirements:

- A timesharing system has a highly unpredictable load, since users continually make different demands on the system. By comparison, a TP load is very regular, running similar load patterns every day.
- Timesharing systems have less stringent availability and atomicity requirements than TP systems. The TP concept of ACID execution doesn't apply.
- Timesharing applications are not mission-critical to the same degree as TP applications and therefore have weaker availability requirements.
- Timesharing system performance is measured in terms of system capacity, such as instructions per second and number of on-line users. Unlike TP, there are no generally accepted benchmarks that accurately represent the behavior of a wide range of timesharing applications.

Client-Server

In a client-server system, a large number of personal computers communicate with shared servers on a local area network. This kind of system is very similar to a TP environment, where a large number of display devices connect to shared servers that run transactions. In some sense, TP systems were the original client-server systems with very simple desktop devices, namely, dumb terminals. As desktop devices have become more powerful, TP systems and personal computer systems have been converging into a single type of computing environment with different kinds of servers, such as file servers, communication servers, and TP servers.

There are many more system types than we have space to include here. Some examples are embedded systems, computer-aided design systems, data streaming systems, electronic switching systems, and traffic control systems.

Why Engineer a TP System?

Each system type that we looked at is designed for certain usage patterns. Although it is engineered for that usage pattern, it actually can be used in other ways. For example, people have used timesharing systems to run TP applications. These applications typically do not scale very well or use operating system resources very efficiently, but it can be done. For example, people have built special-purpose TP systems using real-time systems, and batch systems to run on a timesharing system.

TP has enough special requirements that it's worth engineering the system for that purpose. The amount of money businesses spend on TP systems justifies the additional engineering work vendors do to tailor their system products for TP—for better performance, reliability, and ease-of-use.

1.8 TP SYSTEM CONFIGURATIONS

When learning the principles of transaction processing, it is helpful to have a feel for the range of systems where these principles are applied. We already saw some examples in Section 1.5 on TP benchmarks. Although those benchmark applications have limited functionality, they nevertheless are meant to be representative of the kind of functionality that is implemented for complete practical applications.

In any given price range, including the very high end, the capabilities of TP applications and systems continually grow, in large part due to the steadily declining cost of computing and communication. These growing capabilities enable businesses to increase the functionality of classical TP applications, such as travel reservations and banking. In addition, every few years, these capabilities enable entirely new categories of businesses. In the past decade, examples include large-scale Internet retailers and social networking web sites.

There is no such thing as an average TP application or system. Rather, systems that implement TP applications come in a wide range of sizes, from single servers to data centers with thousands of machines. And the applications themselves exhibit a wide range of complexity, from a single database with few dozen transaction types to thousands of databases running hundreds of millions of lines of code. Therefore, whatever one might say about typical TP installations will apply only to a small fraction of them and will likely be outdated within a few years.

A low-end system could be a departmental application supporting a small number of users who perform a common function. Such an application might run comfortably on a single server machine. For example, the sales and marketing team of a small company might use a TP application to capture sales orders, record customer responses to sales campaigns, alert sales people when product support agreements need to be renewed, and track the steps in resolving customer complaints. Even though the load on the system is rather light, the application might require hundreds of transaction types to support many different business functions.

By contrast, the workload of a large Internet service might require thousands of server machines. This is typical for large-scale on-line shopping, financial services, travel services, multimedia services (e.g., sharing of music, photos, and videos), and social networking. To ensure the service is available 24 hours a day, 7 days a week (a.k.a. 24×7), it often is supported by multiple geographically distributed data centers. Thus if one data center fails, others can pick up its load.

Like hardware configuration, software configurations cover a wide range. The system software used to operate a TP system may be proprietary or open source. It may use the latest system software products or ones that were introduced decades ago. It may only include a SQL database system and web server, or it may include several layers of transactional middleware and specialized database software.

The range of technical issues that need to be addressed is largely independent of the hardware or software configuration that is chosen. These issues include selecting a programming model; ensuring the ACID properties; and maximizing availability, scalability, manageability, and performance. These issues are the main subject of this book.

1.9 SUMMARY

A **transaction** is the execution of a program that performs an administrative function by accessing a shared database. Transactions can execute on-line, while a user is waiting, or off-line (in batch mode) if the execution takes longer than a user can wait for results. The end-user requests the execution of a transaction program by sending a request message.

A transaction processing application is a collection of transaction programs designed to automate a given business activity. A TP application consists of a relatively small number of predefined types of transaction programs. TP applications can run on a wide range of computer sizes and may be centralized or distributed, running on local area or wide area networks. TP applications are mapped to a specially engineered hardware and software environment called a TP system.

The three parts of a TP application correspond to the three major functions of a TP system:

1. Obtain input from a display or special device and construct a request.
2. Accept a request message and call the correct transaction program.
3. Execute the transaction program to complete the work required by the request.

Database management plays a significant role in a TP system. Transactional middleware components supply functions to help get the best price/performance out of a TP system and provide a structure in which TP applications execute.

There are four critical properties of a transaction: atomicity, consistency, isolation, and durability. Consistency is the responsibility of the program. The remaining three properties are the responsibility of the TP system.

- Atomicity: Each transaction performs all its operations or none of them. Successful transactions commit; failed transactions abort. Commit makes database changes permanent; abort undoes or erases database changes.
- Consistency: Each transaction is programmed to preserve database consistency.
- Isolation: Each transaction executes as if it were running alone. That is, the effect of running a set of transactions is the same as running them one at a time. This behavior is called serializability and usually is implemented by locking.
- Durability: The result of a committed transaction is guaranteed to be on stable storage, that is, one that survives power failures and operating system failures, such as a magnetic or solid-state disk.

If a transaction updates multiple databases or resource managers, then the two-phase commit protocol is required. In phase one, it ensures all resource managers have saved the transaction's updates to stable storage. If phase one succeeds, then phase two tells all resource managers to commit. This ensures atomicity, that is, that the transaction commits at all resource managers or aborts at all of them. Two-phase commit usually is implemented by a transaction manager, which tracks which resource managers are accessed by each transaction and runs the two-phase commit protocol.

Performance is a critical aspect of TP. A TP system must scale up to run many transactions per time unit, while giving one- or two-second response time. The standard measures of performance are the TPC benchmarks, which compare TP systems based on their maximum transaction rate and price per transaction for a standardized application workload.

A TP system is often critical to proper functioning of the enterprise that uses it. Therefore, another important property of TP systems is availability; that is, the fraction of time the system is running and able to do work. Availability is determined by how frequently a TP system fails and how quickly it can recover from failures.

TP systems have rather different characteristics than batch, real-time, and data warehouse systems. They therefore require specialized implementations that are tuned to the purpose. These techniques are the main subject of this book.