Chapter 1

What Is It All About?

"If I had had more time, I could have written you a shorter letter." (Blaise Pascal)

1.1 Goal and Overview of the Chapter

Transaction processing is an important topic in database and information systems. Moreover, it is rapidly gaining importance outside the context in which it was originally developed. In this introductory chapter, we discuss why transactions are a good idea, why transactions form a reasonable abstraction concept for certain classes of real-life data-management and related problems, as well as what can and what cannot be done with the transaction concept.

application areas

The transaction concept was originally developed in the context of *database management systems* as a paradigm for dealing with concurrent accesses to a shared database and for handling failures. Therefore, we start out (in Section 1.2) by describing typical application scenarios for database and other information systems in which transactions make sense. The original and most "canonical" application example is funds transfer in banking; very similar applications in terms of functionality and structure have arisen in a number of other service-providing industries, most notably in the travel industry with their flight, car, and hotel bookings. All these classical application examples are commonly referred to as *online transaction processing* or *OLTP* for short. In addition, we will see that the application area of the transaction concept includes modern business sectors such as *electronic commerce* and the management of *workflows* (which are also known as "business processes").

In terms of the underlying computer and network infrastructure, we are typically dealing with distributed systems of potentially large scale and with possibly heterogeneous, interoperating components. Most often, one of these components is a

database management system or, more specifically, a *database server* that processes requests issued by clients (workstations, personal computers, portable notebooks, PDAs, electronic sensors, and other embedded systems, etc.). It turns out that in today's diverse information-technology landscapes, mail servers, Web- or Intranet-based document servers, and workflow management systems also play an increasingly important role and call for transactional support.

transaction concept

The key problem that the transaction concept solves in a very elegant way is to cope with the subtle and often difficult issues of keeping data consistent even in the presence of highly concurrent data accesses and despite all sorts of failures. An additional key property of transactions is that this is achieved in a generic way that is essentially invisible to the application logic (and to application development), so that application developers are completely freed from the burden of dealing with such system issues. This is why transactions are an *abstraction concept*, and why this concept is a cornerstone of modern information technology. Section 1.3 will discuss the role of the transaction concept in state-of-the-art information systems from a "strategic" viewpoint. We will introduce a fairly general reference architecture as a bird's eye view of the entire infrastructure that is necessary to implement and deploy an information system, and we will discuss several variations of this reference architecture that are commonly used in practice. In particular, we will identify components that are in charge of managing persistent data under a transaction-oriented access regime, and we will concentrate on these *transactional (data) servers*. We will then discuss, in Section 1.4, the abstract properties that constitute the transaction concept and the great benefit that these properties provide in the context of a transactional data server. We will also outline the requirements on the server's algorithms in terms of *correctness* and *performance*, as well as of *reliability* and *availability*.

computational models

By far the most important concrete "instantiation" of a transactional data server is a database system. However, this is not a book about database systems. We limit our discussion to topics that are directly and closely related to transactions, and nothing else. We will briefly survey the kind of knowledge we expect our readers to have about database systems in Section 1.5. This will prepare the setting for the introduction of two computational models for transactional servers in the next chapter.

This chapter, like all subsequent chapters, is wrapped up by summarizing, in Section 1.6, the key insights that the reader should have obtained from reading it.

1.2 Application Examples

We begin our exposition with a few examples of applications in which transactional properties can be brought to bear; these scenarios are:

from OLTP to E-commerce and workflow

1. funds transfer in a banking environment, a classical OLTP application,

2. Web-based electronic commerce (E-commerce), and

3. travel planning as a workflow example.

1.2.1 Online Transaction Processing: Debit/Credit Example

Consider the simplified operation of a bank that uses a relational database for keeping track of its account business. The database contains, among others, a table named *Account* which describes bank accounts in terms of their account id, associated customer name, identification of the respective bank branch, and balance. Transactions in the bank are either *withdrawals* or *deposits* (which is why the application is often characterized as consisting of "debit/credit transactions"), which are often combined in *funds transfers.* The typical structure of a debit/credit program is shown below, using commands of the standardized database query language SQL and embedding these commands in a C program. Note the distinction between local variables of the invoked program and the data in the underlying database that is shared by all programs. Also note that a realistic, full-fledged debit/credit program may include various sanity checks against the account data (e.g., for high amounts of withdrawals), in between the SQL Select command and the subsequent Update step.


```
/* debit/credit program */
void main()
{
      EXEC SQL BEGIN DECLARE SECTION;
            int accountid, amount; /* input variables */
            int balance; /* intermediate variable */
      EXEC SQL END DECLARE SECTION;
/* read user input */
      printf("Enter Account ID, Amount for deposit (positive) or withdrawal (negative):");
      scanf("%d %d", &accountid, &amount);
/* determine current balance of the account,
   reading it into a local, intermediate, variable of the program */
      EXEC SQL Select Account_Balance Into :balance
            From Account
            Where Account_Id = :accountid;
/* add amount (negative for withdrawal) */
balance = balance + amount;
/* update account balance in the database */
      EXEC SQL Update Account
            Set Account_Balance = balance
            Where Account_Id = :accountid;
      EXEC SQL Commit Work;
}
```

We may assume the bank operates in a traditional way, with various tellers at its local branches executing transactions. However, the role of a teller's terminal could also be replaced by a customer's PC equipped with home-banking software, the credit-card reader of a merchant, or some other form of smart "cybercash" carrier. We will generally refer to these as "clients" and will disregard the details of the client software, as these are irrelevant to the server on which the database resides.

concurrency and parallelism → concurrency control techniques for isolation

With a huge number of clients potentially issuing simultaneous requests to the bank's database server, *concurrent* (i.e., overlapping in time) or even *parallel* (i.e., on multiple processors) execution of multiple debit/credit transactions is mandatory in order to exploit the server's hardware resources. For example, while the server is waiting for the completion of a disk I/O on behalf of one transaction, its CPU should be utilized to process another transaction; similarly, multiple transactions should be processed in parallel on a multiprocessor machine. Thus, the "outside world" of an individual transaction is continuously changing as concurrently executing transactions are modifying the underlying database of the banking application. In order to be able to ignore the potential fallacies of this concurrency, we would therefore wish that each transaction be executed in an *isolated* manner, i.e., as if there were no other transactions and hence no concurrency. We will see that this tension between concurrency for the sake of performance, on the one hand, and potential sequential execution for the sake of simplicity and correctness on the other, is reconciled by the *concurrency control techniques* of a transactional server.

The following scenario illustrates that concurrency is indeed more tricky than it may seem at first glance, and that it may have a disastrous impact on the consistency of the underlying data and thus the quality of the entire information system even if each individual transaction is perfectly correct and preserves data consistency.

Example 1.1

Consider two debit/credit transactions (i.e., invocations of the debit/credit program outlined above) that are concurrently executed by processes P_1 and P_2, respectively, both operating on the same account x (i.e., the value of the program's input variable "accountid"). To distinguish the two different instances of the local program variable "balance" that temporarily holds the value of the account balance, we refer to them as "balance1" for process $P1$ and "balance2" for $P2$. For simplicity, we ignore some syntactic details of the embedded SQL commands. The first transaction intends to withdraw \$30, and the second transaction intends to deposit \$20. We assume that the initial account balance is \$100. The table below shows those parts of the two transactions that read and modify the account record.

P_1	Time	P_2
/ balance1 = 0, x.Account_Balance = 100, balance2 = 0 */*		
Select Account_Balance Into :balance1 From Account Where Account_Id = x	1	
/ balance1 = 100, x.Account_Balance = 100, balance2 = 0 */*		
	2	Select Account_Balance Into :balance2 From Account Where Account_Id = x
/ balance1 = 100, x.Account_Balance = 100, balance2 = 100 */*		
balance1 = balance1 - 30	3	
/ balance1 = 70, x.Account_Balance = 100, balance2 = 100 */*		
	4	balance2 = balance2 + 20
/ balance1 = 70, x.Account_Balance = 100, balance2 = 120 */*		
Update Account Set Account_Balance = :balance Where Account_Id = x	5	
/ balance1 = 70, x.Account_Balance = 70, balance2 = 120 */*		
	6	Update Account Set Account_Balance = :balance2 Where Account_Id = x
/ balance1 = 70, x.Account_Balance = 120, balance2 = 120 */*		

So upon completion of the execution, the balance of account x, as recorded in the persistent database, will be \$120, although it should be \$90 after execution of the two transactions. Thus, the recorded data does no longer reflect reality and should be considered incorrect. Obviously, this kind of "anomaly" must be prevented by all means, for such an information system to be meaningful and practically viable. Thus concurrent executions must be treated with extreme care. Similar anomalies could arise from failures of processes or entire computers during the execution of a transaction, and need to be addressed as well. □

failures
→ recovery techniques for atomicity and durability

A second fundamentally important point is that the various accesses a transaction has to perform need to occur *in conjunction*. In other words, once a transaction has started, its data accesses should look to the outside world as an atomic operation which is either executed completely or not at all. This property of *atomicity* will turn out to be a crucial requirement on database transactions. Moreover, this conceptual property should be guaranteed to hold even in a failure-prone environment where individual processes or the entire database server may fail at an arbitrarily inconvenient point in time. To this end, a transactional server provides *recovery techniques* to cope with failures. In addition to ensuring transaction atomicity, these techniques also serve to ensure the *durability* of a transaction's effects once the transaction is completed.

The following scenario illustrates that atomicity is a crucial requirement for being able to cope with failures.

Example 1.2

Consider the following *funds transfer* program, which transfers a given amount of money between two accounts, by first withdrawing it from a source account and then depositing it in a target account. The program is described in terms of SQL statements embedded into a "host program" written in C.

```
/* funds transfer program */
void main()
{
       EXEC SQL BEGIN DECLARE SECTION;
             int sourceid, targetid, amount; /* input variables */
       EXEC SQL END DECLARE SECTION;
/* read user input */
       printf("Enter Source ID, Target ID, Amount to be transfered:");
       scanf("%d %d %d", &sourceid, &targetid, &amount);
/* subtract desired amount from source */
       EXEC SQL Update Account
             Set Account_Balance = Account_Balance - :amount
             Where Account_Id = :sourceid;
/* add desired amount to target */
       EXEC SQL Update Account
             Set Account_Balance = Account_Balance + :amount
             Where Account_Id = :targetid;
       EXEC SQL Commit Work;
}
```

Now assume that the above funds-transfer program has started executing and has already performed its first update statement, withdrawing the specified amount of money from the source. If there is a computer hardware or software failure that interrupts the program's execution at this critical point, the remaining second part will not be performed anymore. Thus, the target account will not receive the money, so that money is effectively lost "in transit".

A recovery procedure, to be invoked after the system is restarted, could try to find out which updates were already made by ongoing transaction program executions and which ones were not yet done, and could try to fix the situation in some way. However, implementing such recovery procedures on a per-application case basis is an extremely difficult task that is itself error-prone by its mere complexity, especially because multiple transactions issued by different programs may have accessed the data at the time of the failure. So rather than programming recovery in an ad-hoc manner for each application separately, a systematic approach is needed.

System-provided recovery that ensures the atomicity of transactions greatly simplifies the understanding of the post-failure state of the data and the overall failure handling on the application side. In the example scenario, rather than being left with the inconsistent state in the middle of the transaction, the system recovery should restore the state as of before the transaction began. On the other hand, if the transaction had already issued its "commit transaction" call and had received a positive return code from the system, then the "all" case of the all-or-nothing paradigm would apply, and the system would henceforth guarantee the durability of the transaction's complete funds transfer. □

The above conceptual properties of a transaction, namely atomicity, durability, and isolation, together provide the key abstraction that allows application developers to disregard concurrency and failures, yet the transactional server guarantees the consistency of the underlying data and ultimately the correctness of the application. In the banking example, this means that no money is ever lost in the "jungle" of electronic funds transfers and customers can perfectly rely on electronic receipts, balance statements, and so on. As we will see in the next two application scenarios, these cornerstones for building highly dependable information systems can be successfully applied outside the scope of OLTP and classical database applications as well.

1.2.2 Electronic Commerce Example

In today's information landscape, client requests may span multiple databases and other information sources across enterprise boundaries, yet the mutual consistency of all this data is crucial and thus important to maintain. Then, the resulting transactions operate in a distributed system that consists of multiple servers, often with heterogeneous software. As a concrete example of such a modern setting consider what happens when a client intends to purchase something from an Internet-based bookstore; such applications are known as *electronic commerce* (E-commerce).

The purchasing activity proceeds in the following steps:

1. The client connects to the book store's server through an appropriate Internet protocol, and starts browsing and querying the store's catalog.

2. The client gradually fills an electronic shopping cart with items that she intends to purchase.

3. When the client is about to check out, she reconsiders the items in her shopping cart and makes a final decision on which items she will purchase.

4. The client provides all the necessary information for placing a definitive (and legally binding) order. This includes her shipping address, and also information on her credit card or some other valid form of cybercash. The latter information may be encrypted such that the merchant can only verify its authenticity, but possibly without being able to actually decrypt the provided data.

5. The merchant's server forwards the payment information to the customer's bank, credit card company, or some other clearinghouse for cybercash. When the payment is accepted by the clearinghouse, the shipping of the ordered items is initiated by the merchant's server and the client is notified on the successful completion of the E-commerce activity.

So why are transactions and their properties relevant for this scenario? It is obviously important to keep certain data consistent, and this data is even distributed across different computers. The consistency requirement is already relevant during the catalog-browsing phase when the user fills her shopping cart, as the client's view of the shopping cart should ideally be kept consistent with the shopping-cart contents as maintained by the merchant's server. Note that this should be satisfied in the presence of temporary failures at the client or the server side (e.g., a software failure of a client or server process) and also network failures (e.g., undelivered messages due to network congestion). Further note that this seemingly simple requirement may transitively involve additional data, say on the inventory for the selected items, that could reside on yet another computer.

While one could argue that data consistency is merely an optional "luxury" feature for the shopping-cart contents and does not necessarily justify the use of advanced technology like transactions in the technical sense of this book, a very similar situation arises in the last step of the entire activity. There, it is absolutely crucial that three parties agree on the data that tracks the overall outcome:

1. The merchant's server must have records on both the order and the successfully certified payment;

2. at the same time the clearinghouse must have a record on the payment, too, as its approval may be requested again later or the clearinghouse may be responsible for the actual money transfer;

3. finally, the client must have received the notification that the ordered items are being shipped.

When these three effects on three different computers are known to be atomic, it greatly increases our confidence in the correct processing of such E-commerce activities. Conversely, when atomicity is not guaranteed, all sorts of complicated cases arise, such as the merchant shipping the items but the clearinghouse losing all records of its cybercash approval and ultimately not being able to reclaim the money. Similarly, when the customer is never informed about the shipping and the resulting money transfer, she may order the items again from a different merchant, ending up with two copies of the same book. Even worse, the customer may receive the shipped items and keep them, but pretend that she never ordered and never received them. Then, it is obviously important for the entire E-commerce industry to rely on atomicity guarantees in order to prove a customer's order when it comes to a law suit.

Similar, yet more involved, arguments can be brought up about isolation properties, but the case for transactions should have been made sufficiently clear at this point. Of course, one could deal with inconsistent data among the three computers of our scenario in many other ways as well. But the decisive point is that by implementing the last step of the activity as a transaction, all the arguments about atomicity in the presence of failures can be factored out and the entire application is greatly simplified.

So this example has indeed much more in common with the debit/credit scenario than one would have thought upon first glance. There are, however, a number of important differences as well, and these nicely highlight the potential generalization of the transaction concept beyond the classical setting of centralized database applications:

- The entire application is *distributed* across multiple computers, and the software may be *heterogeneous* in that different database systems are used at the various servers. (Of course, the hardware is likely to be heterogeneous, too, but this is mostly masked by the software and thus less relevant.)

- The servers are not necessarily based on database systems; they may as well be some other form of *information repository* or *document management servers* in general.

- The effects of a transaction may even include *messages* between computers, for example, the notification of the customer. So transactions are not limited to what we usually perceive as "stored data".

We will see in this book that transaction technology can cope well with all these additional, challenging aspects of modern applications. It will take us to some of the advanced material, however, to cover all issues.

1.2.3 Workflow Management: Travel Planning Example

A final and most challenging application class that we consider is so-called *workflows*, also known as (the computerized part of) *business processes*. A workflow is a set of *activities* (or steps) that belong together in order to achieve a certain business goal. Typical examples would be the processing of a credit request or insurance claim in a bank or insurance company, respectively, the work of a program committee for a scientific conference (submissions, reviews, notifications, etc.), the administrative procedures for real estate purchase, or the "routing" of a patient in a hospital. To orchestrate such processes, it is obviously crucial to specify (at least a template for) the control flow and the data flow between activities, although it may still be necessary to improvise at run-time (e.g., in medical applications). Making the "flow of work" between activities explicit in that it is factored out of the entire application is exactly the key leverage from workflow technology in that it allows a company or other institution to largely automate the repetitive, stereotypic parts of its processes while retaining flexibility, and to quickly adjust these processes to changing business needs.

Activities can be completely automated or based on interaction with a human user and intellectual decision-making. This implies that workflows can be long-lived up to several days or weeks, or even months and years. A typical characteristic of workflows is that the activities are distributed across different responsible persons and different, independent information systems, possibly across different enterprises. In particular, an activity can spawn requests to an arbitrary "invoked application" that is provided by some server independently of the current workflow. Thus, workflow management is essentially an umbrella for the activities and invoked applications that constitute a particular workflow. To this end, a workflow management system provides a specification environment for registering activities and specifying, in a high-level declarative way, the control and data flow within a process, and a run-time environment that automatically triggers activities according to the specified flow. Workflow management systems with such capabilities are commercially available and are gaining significant industrial relevance.

As a concrete example for a workflow, consider the activities that are necessary in the planning of a business trip, say, a trip to a conference. Suppose your manager (or professor) allows you to choose one scientific or developer's conference that you can attend as part of a continuing-education program. This involves the following activities:

- Select a conference, based on its subject, technical programme, time and place. If no suitable conference is found, then the process is terminated.

- Check out the cost of the trip to this conference, typically by delegation to a travel agency.

- Check out the registration fee for the conference, which often depends on your memberships, tutorials that you may wish to attend, etc.

- Compare the total cost of attending the selected conference to the allowed budget, and decide to attend the conference only if the cost is within the budget.

With the increasing costs of conferences and ever tighter travel budgets (at the time when this book was written), it is desirable to allow several trials with different conferences, but the number of trials should better be limited, in order to guarantee termination of the entire process. The activities and the control flow between them are graphically depicted in Figure 1.1. This illustration is based on a specification formalism that is known as *statecharts*, which is one particular kind of formal specification method that might be used by a workflow management system. Each oval denotes a state in which the workflow can be during its execution. Each state in turn corresponds to one activity; so the activity is spawned when the state is entered. The activity may then invoke further application programs. When the workflow is started, a specified initial state, i.e., a state without predecessors, is entered, and the workflow terminates when a final state, i.e., a state without successors, is reached. In the example, the initial state is the "SelectConference" state, and the final states are "Go" and "No".

The transitions between states are governed by event-condition-action rules that are attached to the transition arcs as labels. The meaning of a rule of the form "E [C] / A" is that the transition fires if event E has occurred and condition C is true in the current state. Then the current state is left and the state where the transition arc points to is entered; during this transition the specified action A is performed. In the example, we only make use of conditions and actions. Both refer to a small set of variables (instantiated for each workflow instance) that are relevant for the control flow. This kind of control flow specification allows conditional execution as well as loops based on high-level predicates. The entire specification can be hierarchical, thus supporting both top-down refinement as well as bottom-up composition of existing building blocks, by allowing states to be nested. So a state can in turn contain another state chart that is executed when the state is entered. In addition, the specification formalism allows parallel execution also, which is graphically indicated by breaking a state down into two or more "orthogonal" statecharts, separated by a dashed line, that are executed in parallel. In the example, the activities that correspond to the two states "CheckConfFee" and "CheckTravelCost" are executed in parallel. These two states are further refined into several steps, where "CheckTravelCost" again leads to two parallel substates.

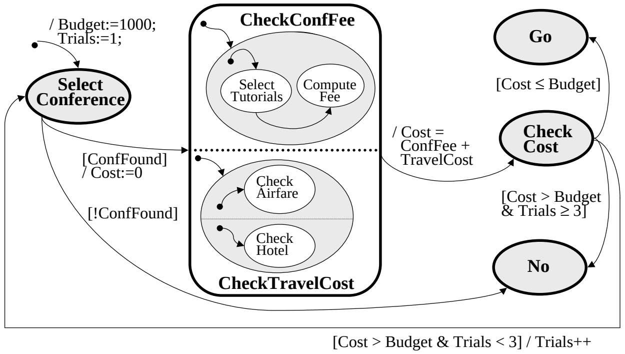

Figure 1.1: Specification of the Travel Planning Workflow.

Although the example scenario is still largely oversimplified, the above discussion already indicates some of the “semantically rich” process-design issues that come along with workflow management. We are here interested in the connection between workflows and transactions, and how a workflow application could possibly benefit from transaction-supporting services. The answer is threefold and involves different stages of “transactional scope”.

1. The activities themselves can, of course, spawn requests to information systems that lead to transactional executions in these systems. This is almost surely the case with the “CheckTravelCost” activity. The travel agency’s invoked application would typically issue transactions against the reservation systems of airlines and hotels. In fact, it seems to make sense that this activity not just figures out the prices, but also makes reservations in the underlying information systems. Obviously, booking a flight to a certain city and a hotel room in that city makes sense only if both reservations are successful. If either of the two is unavailable, the whole trip does no longer make sense. So these two steps need to be tied together in a single transaction. Note that this transaction is a distributed one that involves two autonomous information systems.

2. The outcome of the above reservations affects the further processing of the workflow. The requests against the various information systems would return status codes that should be stored in variables of the workflow and would be relevant for the future control flow. For example, not being able to make one of the two necessary reservations in the selected city should trigger going back to the initial “SelectConference” state for another trial. (This is not shown in Figure 1.1 to keep the example specification simple.) Thus, it is desirable (if not mandatory) that the modification of the workflow’s variables be embedded in the same transaction that accesses the airline and hotel databases. In other words, the *state of the workflow application* should be under transactional control as well. This is an entirely new aspect that did not arise in the banking and E-commerce examples. It is questionable if today’s commercial workflow management systems can cope with this issue in the outlined, transactional way. But as we will see, transactional technology does provide solutions for incorporating application state into atomic processing units as well.

3. One could discuss whether the entire travel-planning workflow should be a single transaction that incorporates all effects on the underlying information systems as well as the state of the workflow application itself. After all, the entire workflow should have an all-or-nothing, atomic effect. Ideas along these lines have indeed been discussed in the research community for quite a few years; however, no breakthrough has come in sight so far. The difficulty lies

in the long-lived nature of workflows and the fact that workflows, like simple transactions, run concurrently. Atomicity is therefore coupled with isolation properties: the atomicity of a workflow would imply that no concurrent workflow could ever "see" any partial effects of it. Regardless of the technical details of how isolation can be implemented at all (to be covered in great depth in this book), one would immediately suspect that maintaining such isolation over a time period of hours, days, or weeks, incurs certain performance problems with regard to the progress of concurrent workflows. For this reason, the straightforward approach of turning an entire workflow into a single transaction is absolutely infeasible.

The discussion of the third item above does not imply, however, that the one-transaction-per-activity approach is the only kind of transactional support for workflows. Consider the situation when all necessary reservations have been made successfully, but later it is found that the total cost including the conference fees is unacceptable and it is decided not to attend any conference at all. Now we hold reservations that may later result in charges to our credit card, unless we intervene. So we better make sure that these reservations are cancelled. One approach could be to extend the workflow specification by additional cancellation activities and the necessary control flow. However, it turns out that cancellation-like activities are fairly common in many business processes, and we may end up spending a lot of time in specifying these kinds of things over and over again. So a better solution would be to generalize the particular case at hand into a more abstract notion of *compensation activities*. Each compensation activity would be tied to one of the regular activities of the workflow in the sense that their combined effect is a neutral one from the workflow application's viewpoint. Of course, we would still have to provide code (or a high-level declarative specification) for each compensation activity itself, but modifications to the workflow's control flow specification are no longer required. Rather we could delegate the appropriate triggering of compensation activities to the workflow management system. The transactional technology that we develop in this book does provide the principal means for coping with compensation issues in the outlined, generic way (as opposed to developing specific solution on a per application basis over and over again).

At this point of the book, the major insight from this discussion is to realize that the scope of transactions is not a priori fixed and limited to stored data, but can be (carefully) extended to incorporate various aspects of information-system applications as well.

1.3 System Paradigms

The three application scenarios of Section 1.2 already provide hints on the various system components that it takes to implement and deploy such applications in their entirety and on putting them together into a complete system architecture. In particular, we have seen that we need to separate *clients*, that is, the computers or terminals from which a human user generates computer work, from the *servers* where data and possibly also executable programs reside in various forms. However, this distinction alone is insufficient for characterizing full-fledged modern information systems. We now introduce a more systematic view of these architectural issues in that we set up a *reference architecture* (or framework) to capture the most typical cases that are used in practice.

1.3.1 Three-tier and Two-Tier Architectures

reference architecture: three-tier system

Our reference architecture is illustrated in Figure 1.2. It captures what is most frequently referred to as a *three-tier architecture* by practitioners. It consists of a set of *clients* (PCs, workstations, notebooks, terminals, digital TV set-top boxes, "intelligent" sensors, etc.) that interact with an *application server* which in turn interacts with a *data server*.

clients: presentation processing

Clients send business-oriented (or goal-oriented) requests to the application server. For example, invoking a debit/credit transaction, starting an E-commerce shopping session, or initiating a travel-planning workflow would be concrete examples for such requests. In modern applications, requests are typically issued from a GUI (Graphical User Interface); likewise, the reply that the application server will send back is often presented in a graphical way, using forms, buttons, charts, etc., or even virtual-reality-style animations. All this presentation processing, for both input and output, is done by the client. Therefore, HTML (Hypertext Markup Language), one of the original cornerstones of the World-Wide-Web, is a particularly attractive basis for presentation, because it merely requires that a Web browser is installed on the client side and thus applies to a large set of clients.

application server

The application server has a repository of *application programs* in executable form, and invokes the proper program that is capable of handling a client request. Both the application programs and the entire application server can be organized in a variety of ways. The programs themselves may be anything from an old-fashioned terminal-oriented COBOL program to a Java applet or some other program that generates, for example, a dynamic page.

application programs and business objects

Since the number of such programs that need to be provided for a full-fledged information-system application can be very large, a good way of structuring this program repository is to organize the programs according to the object-oriented

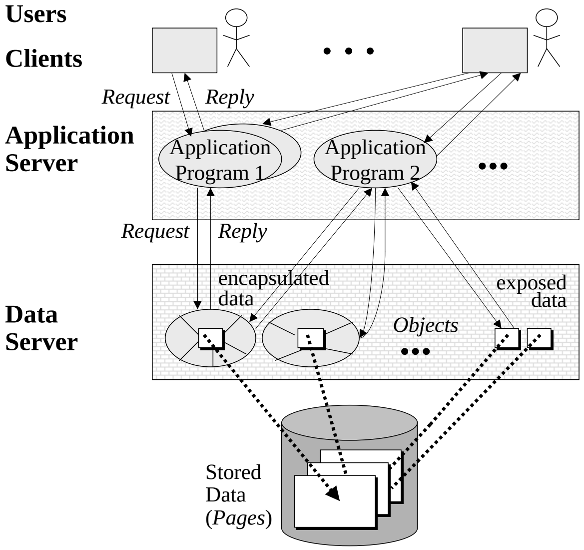

Figure 1.2: Reference Architecture.

paradigm. This means that abstract *business objects* such as accounts, customers, shopping catalogs, etc., are provided as encapsulated objects, each with an abstract interface that consists of a number of *methods* that can be invoked on such an object. The invokable methods are exactly the application programs that refer to (and are thus centered around) a business object. In programming language terms, a business object would be referred to as an *abstract data type (ADT)*. The implementation of these ADTs, as provided by the corresponding application programs, may itself invoke methods on other business objects and issue requests to other servers, particularly, data servers. To give an application example, modern ERP (Enterprise Resource Planning) systems like SAP R/3 largely follow this paradigm.

request brokering

The application server manages the entirety of application programs or business objects in that it spawns execution *threads* on behalf of client requests, monitors executions, handles certain generic forms of exceptions, and so on. It thus constitutes the surrounding run-time system of the invoked programs. The functionality of this run-time system usually includes also the management of the communication connections between the clients and the application server itself. Often some form of *session* is established between a client and the server, possibly on top of a session-less protocol such as the HTTP protocol (Hypertext Transport Protocol) of the World-Wide-Web, and the application server is again in charge of creating, monitoring, and terminating these sessions. So, in essence, the application server can be viewed as a *request broker* that establishes and maintains the proper connections between the client requests and the invoked business-object methods (or application programs in more classical terminology). Traditionally, so-called *TP monitors* (Transaction-Processing monitors) have been the commercial incarnation of such request brokers, with specific support for OLTP applications. In recent years, so-called *object request brokers (ORBs)* have become popular in this role, based on either the standardized CORBA (Common Object Request Broker Architecture) model of distributed objects or the industry de-facto standards DCOM (Distributed Component Object Model) or EJB (Enterprise Java Beans). Yet another category of request brokers are *Web servers* (occasionally referred to as Internet or Intranet information/application servers), which include the functions (and may have grown out) of a mere HTTP server but are significantly enhanced to manage program-execution threads, sessions, etc. As a matter of fact, the boundaries between these three product categories are becoming more and more blurred, with TP monitors becoming object-oriented or Web servers being "upgraded" in terms of functionality and scalability. It is not unlikely that in a short while only one blend of request broker will exist and be worth remembering. Especially the proliferation of the XML (Extensible Markup Language) data exchange standard will be a major force towards unified and also simpler protocols for network-centric applications. For exactly this reason we refer to the abstract notion of a request-brokering application server only throughout this book. This notion would also include a complete workflow management system that has its own integrated request broker or is coupled with an ORB or TP monitor.

data servers: database servers, document servers, mail servers, and others

As mentioned above, the implementation of a business-object method often involves issuing requests to other business objects or other servers. Most importantly, the business objects themselves typically do not contain any persistent state data that would live beyond request/reply boundaries and user sessions for an indefinite time period. Rather such persistent data is better kept on a *data server* that is specifically geared for the tasks of reliable long-term maintenance of data. *Database systems* are the most prominent type of systems that fall into this category, but for certain types of data, other systems may be even more appropriate. For example, for *semistructured* (text or multimedia) documents and for electronic mail, specific products have a very successful usage record. Thus, *document servers* and *mail servers* are other important types of data servers. All these data servers may also provide some notion of encapsulated objects, as opposed to exposing the "raw" data at their interface. However, this is an option that does not need to be exercised. For example, the data of a database server may be accessible to the application programs of an application server via the standardized query language SQL (Structured Query Language) in a direct way or via stored procedures, user-defined ADT functions, or other forms of encapsulated interfaces in modern object-relational or fully object-oriented database products. In all cases, the actual data items ultimately reside in fixed-size containers commonly known as *database pages* (or *"blocks"*) on secondary storage, that is, the server's (magnetic) disks.

specializations of the reference architecture: two-tier systems

Most of the elements in the above discussion are illustrated in Figure 1.2. In addition, the figure implicitly includes a number of practically important specializations that are obtained by collapsing two tiers into a single component. The two major options for such simpler architectures are:

- Combining the client and the application-server tiers: This implies that the application programs reside on the clients, leading to an architecture that is often called a *client-server system with "fat" clients*. Clients then communicate directly with a data server, for example, via SQL, often embedded in a standardized client-server high-level communication protocol such as ODBC (Open Database Connectivity). A possible problem with this approach (and one of the original motivations for the introduction of an explicit "middle tier") is that many data-server products have traditionally lacked the capabilities of maintaining a very large number of sessions and execution threads, say, ten thousand concurrent sessions, which would not be that unusual for popular information providers on the Web. However, not every application really needs such very high degrees of concurrency, and many commercial servers have made tremendous progress in this direction anyway.

- Combining the application-server and the data-server tiers: This implies that the application programs reside in the data server, leading to an architecture that is known as a *client-server system with "thin" clients*. For example, a database system that has rich object-encapsulation capabilities could provide business-object methods to which the client requests can be directly mapped. The potential drawback of this architectural approach is that the data server may become less scalable and more susceptible to being overloaded. Indeed, if this architecture were the starting point, then "outsourcing" the application-processing load to a separate application server would be another motivation for a three-tier architecture. For example, the SAP R/3 product for comprehensive and integrated business administration originally evolved along this "outsourcing" path. However, with today's extremely powerful hardware, the specialized two-tier approach using a combined data/application server is a viable option for many low-end and mid-sized applications.

So the bottom line is that specialized two-tier architectures are and will continue to be practically relevant as well.

To avoid becoming dependent on specific architectures, our approach in this book will be based on *computational models* (to be introduced in Chapter 2) rather than two- or three-tier system architectures. This will allow us to abstract from the particularities of the specific architecture and derive results that are as general as possible. However, for concrete illustration purposes, readers may occasionally find it helpful to transfer general results back to a specific system architecture. For this purpose, the two-tier architectures are typically easier to match with the abstract computational models, and we will therefore prefer also the two-tier cases over the more general three-tier case whenever we need such illustration.

1.3.2 Federations of Servers

federated architecture

It would be wishful thinking to assume that the entire information processing of a large enterprise can be concentrated in a single application server and a single data server. Instead, multiple instances of two- and three-tier architectures often exist simultaneously within an enterprise for the different business units and worldwide branches. Adding the various liaisons with external business partners and the general trend to "virtual enterprises" increases the multitude of information services upon which modern business is based. So we really have to cope with a *highly distributed architecture* consisting of a wide variety of application and data servers. Obviously these servers can be *highly heterogeneous* in that they use different products, different interfaces, and even radically different design philosophies (e.g., classically relational versus object-oriented database systems). In addition, the various servers can differ

distributed, heterogeneous, autonomous servers widely in terms of their "autonomy": some servers may focus explicitly on the specific workload on behalf of a specific business purpose and specific clients, whereas less autonomous servers may be more willing and prepared to interoperate with other application and/or data servers. So the full spectrum in this highly diverse information landscape is best characterized as a *federated system architecture*, with multiple servers working together on a per-business-case basis. This most general kind of architecture, which is expected to become ubiquitous in our information society, is illustrated in Figure 1.3.

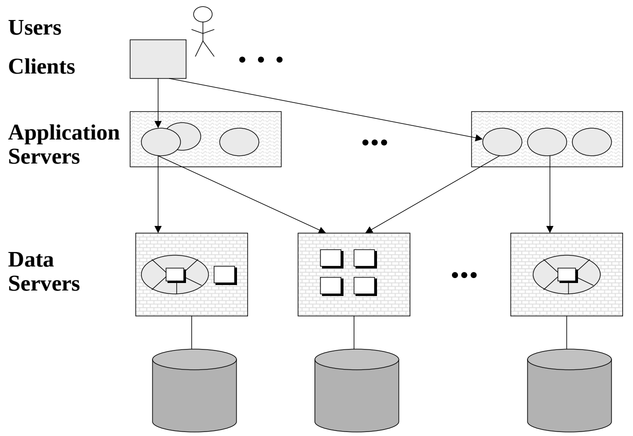

Figure 1.3: Federated System Architecture.

Federated system architectures require a communication infrastructure that can cope well with the heterogeneity and different operation modes of the underlying application and data servers. This infrastructure is often referred to as *middleware*, as its place is between computer operating systems and the application programming level, but it may as well be viewed as an extension of modern operating systems. *middleware*

The key purpose is to provide high-level communication mechanisms between clients and servers and also between servers, reconciling programming comfort (e.g., location independence) with robustness (e.g., reliable sessions). The paradigm that has evolved along these lines is again an object-oriented one: today's middleware is centered around the concept of *remote method invocation*. That is, methods on remote objects can be invoked as if these were objects in the same local address space; all necessary communication including the lookup of computer network addresses is automatically plugged in by the middleware layer. To make this work in a highly heterogeneous software landscape, one obviously needs standardized interfaces. The most prevalent standards to this end are DCOM (Distributed Component Object Model) and CORBA (Common Object Request Broker Architecture); Java-based services are of increasing importance as well.

As mentioned earlier, both DCOM and CORBA services provide a lot more than just bare communication. In particular, both include request brokering facilities, one of the key features of an application server. Consequently, one cannot precisely tell where the middleware stops and the application server itself begins. But this fuzziness is only an issue in terms of implementation. The overall system architecture can still be viewed as if the application server and the middleware between servers and clients were clearly separated. In fact, it was the very intention of this section to condense the often confusing manifold of commercial system components into a sufficiently simple architectural model. From now on, our framework will be set up by the introduced reference model, and we will leave the possible mappings of this model onto commercial software packages to the reader's (past or future) professional experience.

1.4 The Virtues of the Transaction Concept

After having set the stage from an architectural point of view, we now turn to the transaction concept itself and discuss its virtues, properties, and programming interface.

1.4.1 Transaction Properties and the Transaction Programming Interface

The three application scenarios of Section 1.2 suggest that the application's view of a transaction is essentially a program execution that reads and modifies one or more data sources, which are managed by database systems or other types of data servers. The key point of a transaction is that it comes with certain system-guaranteed properties which greatly simplify the development of such applications in that the application programs themselves can safely ignore a good portion of the complexity of the overall system. In particular, application programs are completely freed up from taking care of the issues of

- *concurrency*, that is, all effects that may result from concurrent or even parallel program executions and especially data accesses, and of
- *failures*, that is, all effects that would result from program executions being interrupted (at "inconvenient" points) because of process or computer failures.

masking concurrency and failures to application development

So concurrency and failures are, to a large extent, masked to the application programs. Therefore, application programs can be developed as if they were to be executed in a strictly sequential manner in a failure-free system environment. Such simplifications are possible because all the necessary steps to cope with concurrency and failures are factored out from the diversity of applications, and are delegated to the generic run-time system of the underlying transactional servers. This clear separation of responsibilities into application-specific functionality and generic run-time services is indeed the main contribution of transactional information systems and the key to their impressive commercial success: being able to ignore concurrency and computer failures in application development is a tremendous gain in terms of programmer and development productivity, with respect to both time and cost.

All this is based on the simple, yet extremely powerful abstraction concept called *transaction*. Transactions form the "interface contract" between an application program and a transactional server in the following sense:

1. The application program simply specifies, during its execution, the boundaries of a transaction, by issuing "begin transaction" and "commit transaction" calls. In applications that use embedded SQL the former is often implicit, interpreting the start of the program execution or its session with the database server as the transaction begin, whereas the latter is explicit and marked by the "Commit Work" call (see the debit/credit and funds transfer examples in Section 1.2). The end of a transaction may also implicitly mark the begin of a new transaction within the same program invocation.
2. The server automatically considers all requests that it receives from the application program within this (dynamic) scope as belonging to the same transaction and, most importantly, guarantees certain properties for this set of requests and their effects on the underlying data.

ACID properties

The properties a server guarantees for a transaction are known as the *ACID properties*, and comprise the following four properties (whose initial letters form the word "acid", hence the name):

atomicity

transaction commit

Atomicity: From the client's and the application program's point of view, a transaction is executed completely or not at all, that is, in an all-or-nothing fashion. So the effects of the executing program on the underlying data servers will only become visible (to the outside world or to other program executions) if and when the transaction reaches its *"commit"* point, i.e., the "commit transaction" call is successfully returned. This case implies that the transaction could be processed completely, and no errors whatsoever were discovered while it was processed. If, on the other hand, the program is abnormally terminated before reaching its commit point, the data in the underlying data servers will be left in or automatically brought back to the state in which it was before the transaction started. So, in this latter case, the data appears as if the transaction had never been invoked at all. This holds also for system failures that are not related to the executing transaction itself, such as failures of the data server, the operating system that surrounds it, or the computer on which the data server is running.

consistency-preservation

Consistency-preservation: Consistency constraints that are defined on the underlying data servers are preserved by a transaction; so a transaction leads from one consistent state to another. For example, the merchant's, the bank's, and the customer's records on an electronic sale should be mutually consistent, or in a hyper-linked document collection, the referential structure should be consistent in that neither dangling references nor unreachable documents exist. Upon the commit of a transaction, constraints like these must be satisfied; however, between the beginning and the end of a transaction, inconsistent intermediate states are tolerated and may even be unavoidable.

transaction abort (rollback) upon consistency violation

This property of consistency-preservation cannot be ensured in a completely automatic manner. Rather, it is necessary that the application is programmed such that the code between the beginning and the commit of a transaction will eventually reach a consistent state. This can be hidden in the application logic or be done by including corresponding code in the application programs; moreover, the task of checking consistency can be transfered to the database system, by exploiting constraint-declaration statements or even triggers. In the latter case, failing to satisfy all specified constraints at the commit point of a transaction results in the data server enforcing an abnormal termination of the transaction. Then, in conjunction with the atomicity property, consistency is preserved. Alternatively, the application program can explicitly notify the data server when it is clear that it will no longer be able to reach a consistent state and requests that it be abnormally terminated. In this case, we say that the transaction is deliberately *aborted*, and the application program can achieve this effect by issuing an explicit *"rollback transaction"* call. Again, in conjunction with the atomicity property, data consistency is preserved. The capability to explicitly request such a transaction abort can be a great simplification of the application programs' consistency-checking and consistency-maintenance code.

isolation

Isolation: A transaction is isolated from other transactions, in the sense that each transaction behaves as if it was operating alone with all resources to itself. In particular, each transaction will "see" only consistent data in the underlying data sources. More specifically, it will see only data modifications that result from committed transactions, and it will see them only in their entirety, and never any effects of an incomplete transaction. Obviously, this is the decisive property that allows us to hide the fallacies and pitfalls of concurrency from the application developers. As we will see later, a sufficient condition for isolation is that concurrent executions are equivalent to sequential ones, so that all transactions appear as if they were executed one after the other rather than in an interleaved manner.

durability (persistence)

Durability: When the application program is notified that a transaction has been successfully completed, i.e., when the "commit transaction" call is successfully returned, all updates that the transaction has made in the underlying data servers are guaranteed to survive subsequent software or hardware failures. Thus, updates of committed transactions are durable (until another transaction later modifies the same data items) in that they persist even across failures of the affected data server(s). Hence the name "durability" or "persistence".

transaction programming interface

In summary, a transaction is a set of operations executed on one or more data servers that are issued by an application program and are guaranteed to have the ACID properties by the run-time system of the involved servers. The ACID contract between the application program and the data servers requires the program to demarcate the boundaries of the transaction as well as the desired outcome – successful or abnormal termination – of the transaction, both in a dynamic manner. To this end, the programming interface of a transactional information system conceptually needs to offer three calls:

- *"begin transaction"* to specify the beginning of a transaction,
- *"commit transaction"* to specify the successful end of a transaction, and
- *"rollback transaction"* to specify the unsuccessful end of a transaction with the request to abort the transaction.

Note, once more, that application programs generate the transaction boundaries as well as the operations that constitute a transaction dynamically during program execution (and, as mentioned, often implicitly for beginning transactions and explicitly for committing them). So, in particular, programs can generate sequences of successive transactions, loops of transaction invocations, or even multiple transactions in parallel, provided that the program is itself multi-threaded. Thus, a workflow management system could be viewed as a multi-threaded application program that issues transactions in such a flexible manner. As pointed out earlier, there is a lot more to workflow management, but it is clear at this point that the transaction concept is a great asset for distributed workflows and a wide variety of application classes beyond the traditional OLTP area.

1.4.2 Requirements on Transactional Servers

The core requirement on a transactional server is that it provides the ACID guarantees for sets of operations that belong to the same transaction issued by an application program. This requires that the server includes at least the following two main components:

server components: concurrency control and recovery for ACID contract

- a *concurrency control* component to guarantee the isolation properties of transactions, for both committed and aborted transactions, and
- a *recovery* component to guarantee the atomicity and durability of transactions.

Note that the server may or may not provide explicit support for consistency-preservation. One type of server may require the application program to do all consistency checking and maintenance itself with implicit support by means of the rollback option, whereas other servers, especially database systems, could enforce consistency by means of logical rules. Such logical rules are, however, not a specific issue of transactional information systems and will therefore not be considered in this book.

performance: throughput and response time

In addition to the ACID contract, a transactional server should meet a number of technical requirements: A transactional data server (which could, and most often will, be a database system) must provide *good performance* with a given hardware/software configuration, or more generally, a good cost/performance ratio when the configuration is not yet fixed. Performance typically refers to the following two metrics:

- *high throughput*, which is defined as the number of successfully processed transactions per time unit, and

- *short response times*, where the response time of a transaction is defined as the time span between issuing the transaction and its successful completion as perceived by the client.

It is evident that these performance metrics depend on many issues in the implementation, the configuration, and the tuning of a server. As far as transactions are concerned, the important requirement is that the concurrency control and recovery components do not become a bottleneck. Therefore, it is desirable that a transactional server is able to execute as many transactions as possible concurrently or in parallel (with the parallel case assuming that the underlying computer is a multiprocessor). Of course, such high degrees of concurrency or parallelism are subject to the condition that the ACID contract is not violated. In fact, it would be much simpler to provide the ACID properties if performance were not an issue. Real applications, however, dictate that throughput and response time are important yardsticks for assessing the concurrency control and recovery components of a transactional server.

reliability

A server must be *reliable*, meaning that it virtually never loses data, unless certain forms of catastrophic failures occur, and that it is always able to recover data to a desired consistent state, no matter what complicated forms of one or multiple failures arise. Reliability alone means correct behavior in the presence of failures, but this may incur outages of the server during which client requests are not served. In addition, most applications require high *availability*, which implies that recovery times after failures are short, and that failures that lead to total outages are infrequent.

availability

Other technical requirements on a data server include security, ease of administration, standards compliance, and so on, but these are not directly related to transactions and will therefore be disregarded in this book.

1.5 Concepts and Architecture of Database Servers

As mentioned earlier, the most important class of data servers are database systems. Throughout this book we assume the reader to be familiar with a number of basics from database systems, and we will discuss and explain in detail only those aspects that refer to transactions, concurrency control, and recovery. In this section, to make the book sufficiently self-contained, we merely sketch a few principles of database system internals, specifically, the layered architecture of modern database systems, the storage layout of data, the access acceleration provided by index structures, and the basics of query and update processing. For other database issues, particularly, the relational, object-oriented, and object-relational data models including the corresponding data definition and manipulation languages such as SQL or OQL, we refer to the relevant literature mentioned in the bibliographical section of this chapter.

1.5.1 Architectural Layers of Database Systems

Virtually all modern database systems have a layered internal architecture on the server side, as illustrated in Figure 1.4. Each layer has an interface that provides services that are invoked by the layers above it to implement their higher-level services. Note, however, that all layers are within the same server and therefore not visible to application programs. The actual code base of a specific database system may not conform to the simple structure of the figure, but all systems have internal components that can be approximately mapped to the layers shown. When the server is itself a *distributed* database system, then all system components and thus all architectural layers are usually installed on all the underlying computers.

multi-threaded processing of requests

When a client request arrives at the server, the server executes code that transforms the request into one or more operations at each of the underlying layers, ultimately arriving at a sequence of disk accesses (unless caching avoids the disk access). To allow the server to exploit parallelism between the CPU and the disk(s), the server makes use of modern operating-system services in that it typically spawns one *thread* (also known as "light-weight process") per active request, as illustrated by the wavy lines in Figure 1.4. This typically takes place within one ("heavy-weight") process whose virtual-memory address space is shared across all threads. The server may actually keep a pool of pre-allocated threads that are dynamically assigned to requests rather than creating and destroying a thread for each individual request. This multi-threaded architecture also facilitates disk I/O parallelism between different requests. In addition, by simply assigning more than one thread per request, the architecture is also geared for CPU and I/O parallelism within a single request execution. This is the standard way of implementing a database system on a "symmetric (shared-memory) multiprocessor (SMP)".

language and interface layer

The *language and interface layer* makes various kinds of interfaces available to the application developer. This layer provides one or more languages for data definition and manipulation, usually in the form of application programming interfaces (APIs). By far the most important language of this kind is *SQL*; the most important standard for embedding SQL in an API that is usable for a variety of programming languages is ODBC (Open Database Connectivity), but other proprietary or language-specific SQL-based interfaces are widely used as well. In addition to the data manipulation capabilities in the narrow sense, SQL is often combined with or enriched by programming language constructs, so that encapsulated "business objects" can be implemented within the database server and their operations can be provided at the server interface (see the discussion in Section 1.3).

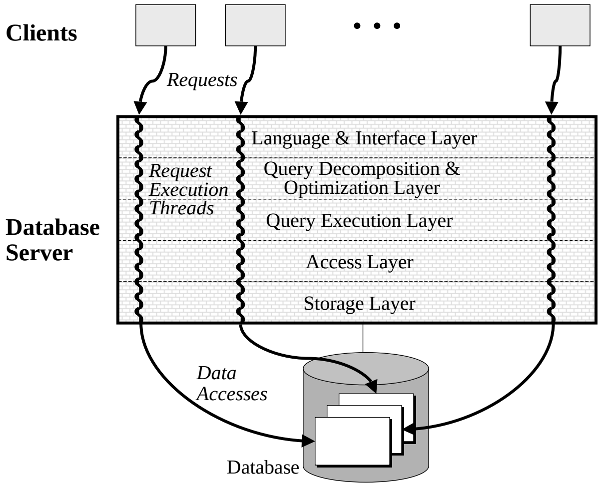

Figure 1.4: Layered Architecture of a Database System.

The language and interface layer itself takes care of the client-server communication at the API level and contains initial processing steps such as parsing the SQL command in a request, authentication and authorization checking, etc. The next lower layer, the *query decomposition and optimization layer*, works on an internal, tree-based representation of a request and is concerned with the further decomposition of the request into smaller units that can be directly executed. In particular, when the request is a full-fledged SQL query with a complex query predicate, this decomposition involves challenging optimization issues. This is the level where the database system makes a choice among different execution plans that are equivalent in that they all implement the same query but can widely differ in terms of their execution efficiency. These optimizations are usually carried out at the application program's compile-time, but advanced applications, such as data warehousing systems, and also a fair number of otherwise simple application programs create the submitted SQL requests dynamically during their run-time and thus force the server to decompose and optimize those requests also at run-time.

query decomposition and optimization layer

The execution plan that is chosen by the query decomposition and optimization layer is usually represented as a tree (or, more generally, a directed acyclic graph) of operators where each operator can be directly mapped to a piece of server code. The code for these operators and the code for the control and data flow among operators is provided by the *query execution layer*. For example, this layer includes the code for sorting disk-resident data, for performing a relational join operation based on hash tables, a join algorithm based on sorting and merging, and so on. All these operators consume streams of data records as input and produce such streams as output. In addition to the data records that constitute the primary contents of the database, the query execution layer also depends heavily on index structures to speed up the processing of the records. An index is essentially a search structure, often a specific type of search tree, that helps identifying records that match a certain predicate (i.e., a search condition). Without indexes, all queries would inevitably end up performing at least one complete scan over a possibly huge collection of data records. Index structures and also the capabilities for accessing and manipulating data records are provided by the *access layer* of a database system. Finally, both data records and index structures are ultimately mapped onto "pages", which are fixed-size storage containers for the transfer between disk and memory (also known as "blocks"). The *storage layer* is responsible for managing the pages of a database. This involves disk I/O control and optimization and, especially, caching in memory to exploit locality in the page access patterns. Usually, the most recently or frequently used pages are kept in memory and can be accessed without disk I/O. The area in main memory that serves this caching purpose is known as the "database (page) cache" or "database (page) buffer".

query execution layer

access layer

storage layer

Note that, a priori, the layered architecture of a database system does not tell us anything about transaction management. The functionality of transactional concurrency control and recovery could indeed be tied to any of the presented five layers. In practice, however, those functions are often integrated into the access and storage layers. We will discuss the embedding of transaction management into a database architecture in more detail when we introduce the book's principal computational models in Chapter 2.

1.5.2 How Data Is Stored

data records

All data objects are internally kept in the form of *data records*. A record consists of a set of *fields* or *columns* and is stored on disk as a string of consecutive bytes. The fields correspond to the attributes of a relational schema; so they are of specific data types, but this is not relevant at the storage level other than possibly requiring specific byte encodings.

pages

Records reside on disk in fixed-size storage containers called *pages*. A typical page size is in the order of 16 KBytes; so with typical record sizes of a few hundred bytes, a page usually contains a number of records. Pages are the minimum unit of data transfer between a disk and main memory, and they are the units of caching in memory. A disk I/O can read or write one or more pages, typically pages that are contiguous on disk. Sometimes a fixed number of pages that can be read or written together is called a "block", but most often, the notion of a block is merely used as a synonym for a page. In this book we will use pages and blocks as synonyms.

storage layout of a page

Each page has a *page header*, typically less than hundred bytes, that contains administrative information for free space management and similar purposes (e.g., the number of free bytes in the page, the size of the largest free area within the page, etc.). Records that contain variable-length columns such as ones of type `varchar` are themselves variable-length strings within a page; so updates to data records can cause growth and shrinkage of these byte strings and may thus require the movement of records within a page. As the database system usually keeps pointers to records in other places, most notably for indexing purposes, it is important to avoid that each and every movement of a record causes those pointers to change. This is achieved by addressing records indirectly through a *slot array* that resides at the end of the page (also known as a "page trailer"). A pointer to a record, a so-called *RID* (record ID, or "row ID"), consists of a page number and a slot number, and it is only this referenced slot inside the page that contains the full byte address of a record. So all record movements are encapsulated within a page, as long as they do not span multiple pages. Sometimes, however, it is unavoidable that a record must move to another page because of lack of space within the original page. In this case, a "forwarding pointer", which is itself a RID, is left in the original page. With this technique, all records are reachable through their RID pointers with at

most two page accesses, and most records are reachable in one page access. This entire storage layout of a page is illustrated in Figure 1.5, using data records that capture information about persons, customers or friends, with fields such as Name, Age, City, Zipcode, Street, etc.

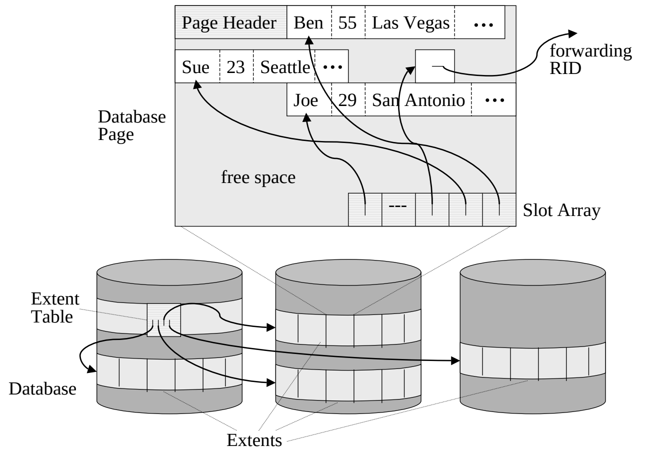

Figure 1.5: Storage Layout of Database Pages.

long fields

A growing number of database applications require support for *long fields*, also known as BLOBs (Binary Large OBjects), to capture multimedia data such as images (e.g., a photograph of a person), text, or video data. Such fields are typically much larger than a page and must therefore be decomposed into multiple chunks and spread across multiple pages. A simple implementation technique for long fields would embed a list of page numbers rather than the actual long field in the data record, and those pages would be exclusively reserved for the chunks of a long field. There exist more sophisticated techniques for the storage layout of long fields, but the simple approach is sufficient for the understanding of this book.

storage layer metadata

In addition to the pages that contain data records, a database system needs to keep a variety of metadata. At the storage level, this mostly refers to the translation of page numbers into physical addresses on disk and to the management of free space within a database. Database systems usually pre-allocate a certain amount of disk space that consists of one or more *"extents"*, each of which is a range of pages that are contiguous on disk. A page number is then translated into a disk number and a physical address on the disk by looking up an entry in a so-called *extent table* and adding a relative offset. Extents and the extent table are illustrated in Figure 1.5. When data records are inserted, the database system gradually uses this pre-allocated space by assigning records to pages and keeping track of the available free space. In particular, all systems maintain some form of *free space management table* from which they can infer which pages are completely empty as well as possibly further free-space information.

As mentioned earlier, the access layer of a database system also keeps certain forms of "secondary" or "derived" data to speed up the searching of records by their contents (as opposed to the access through a given RID). Most notably, this secondary data includes *index structures*, which are the subject of the next subsection. Here we merely point out that those structures are themselves mapped onto pages that are subject to caching, free space management, and other "bookkeeping" at the underlying storage layer. Those pages are referred to as index pages, but the storage layer often treats them in the same way as regular data pages.

1.5.3 How Data Is Accessed

table scan vs. index lookup

Database systems often need to retrieve all data records (of the same table) that have a specified value in a specific field, say, all records with the value "Miami" in their City field. This could be a complete, relatively simple query or merely one step in the processing of a more complex query. The specified value (e.g., "Miami") is referred to as a *search key*; sometimes this term is also used for the name of the field on which the search is performed (i.e., City field in the example). Without any index structures the only way to retrieve the records with the given search key is through a sequential *table scan* in which all records of the corresponding table are accessed and inspected. The typical implementation of a table scan first looks up, in the free space management table, which (non-empty) pages are assigned to the corresponding data table and then accesses all those pages one by one; the I/O cost of this procedure is linear in the size of the data table. In contrast, a more efficient implementation that we would hope for should retrieve the desired data records with an I/O cost that is at most *logarithmic* in the size of the data table (plus linear in the number of the records that satisfy the search condition, or under certain circumstances the number of pages on which these records reside). This is feasible in conjunction with an index structure that keeps all values of a given field (or combination of fields) that occur in the database, along with pointers to those

data records that contain the corresponding value. Using an index structure for resolving a specified search condition is called an *index lookup*.

search trees

The index structure itself can be implemented in a variety of ways, most notably as a *search tree* or a *hash table*, both in a page-structured, disk-resident form. Regardless of the implementation, the interface of the index structure provides a function like

lookup <index> where <indexed field> = <search key>.

With a tree-based implementation, which implicitly captures the ordering of search keys, an additional interface function that cannot be easily supported by hash-based implementations would be

lookup <index> where
<indexed field> between <lower bound> and <upper bound>,

where one of the two bounds could be omitted and is then implicitly set to the lowest or highest possible value of the field's domain (e.g., zero or infinity in the case of non-negative integers). For example, we could retrieve all records with values in the Age field that lie between 20 and 30. This type of index lookup is called a *range lookup* (or range scan), as opposed to the first type that is known as an *exact-match lookup*. Sometimes both types together are referred to as *index scans*. Both return RID lists for the qualifying records.

B^+-trees

In the following we restrict ourselves to tree-based implementations as the more versatile ones. The prevalent data structure in this category is the *B^+-tree*. This is a page-structured search tree whose nodes correspond to pages and where all leaf nodes are at the same tree level (i.e., have the same distance from the root node). An example B^+-tree for an index on the Name field of a Person table is shown in Figure 1.6. This tree has height 2; it consists of a root node and three leaf nodes at the next lower level. The leaf nodes contain all values of the indexed field that exist in the database. Each value is followed by a list of RIDs to the corresponding records. The non-leaf nodes of the index consist of pairs of values and pointers to index nodes of the next lower level. Within a node these entries are sorted by values. These values serve as routers to guide a top-down search for a given key value. To this end, the value that is associated with a pointer to a subtree must be larger than or equal to the highest existing key in this subtree and it must be smaller than the lowest existing key in the right neighboring subtree. For example, the entry in the root of Figure 1.6 that points to the leftmost leaf has the value "Bob", which is the (lexicographically) largest key value in that subtree. Thus, the values in non-leaf nodes effectively partition the key values into disjoint value ranges.

The rightmost child node of a given non-leaf node is automatically separated

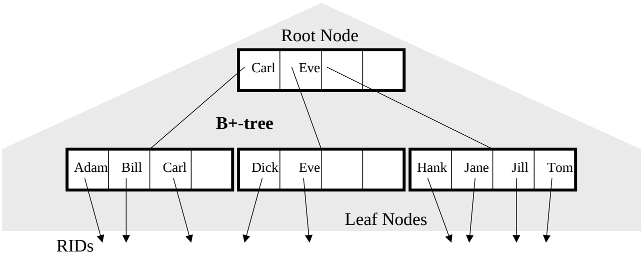

Figure 1.6: Example of a B$^+$-tree.

from its left neighbor by the router value for this neighbor. Therefore, an entry that points to the rightmost child of a node does not have a value but merely consists of a pointer. In the example of Figure 1.6, the third entry of the root simply points to the rightmost leaf, meaning that all key values in this rightmost subtree are higher than the router value "Eve" in the preceding entry of the root. Note that this technique is applicable not only to the root, but avoids an explicit router value for the rightmost subtree among the children of the same node at all levels. The highest possible key value within such a rightmost subtree is then determined by the router value for all subtrees with the same parent, which can be found in that parent's parent. As a consequence, each non-leaf node has one more pointer to the next index level than router values.

A search for a given key value starts at the root node. It determines within the currently inspected node the smallest value that is larger than or equal to the given search key, and then follows the pointer of this router value. If the search key is higher than all values in the index node, then the search follows the pointer to the rightmost child. This step per node is recursively applied until the leaf level is reached. In the example tree, a search for the key "Dick" would follow the "Eve" router in the root and find the key in the middle leaf. In contrast, a search for the key "Joe" would follow the pointer to the rightmost leaf, where it would detect that this key does not exist in the database.

B$^+$-trees are among the most fundamental concepts in database systems. Moreover, they also pose significant challenges for the concurrency control and recovery

components of the database system, as an inconsistency of a B$^+$-tree index (e.g., dangling pointers or a violation of the tree invariant) would be disastrous, and it is crucial to preserve consistency with extremely little overhead. Because of the importance of index structures, we will later devote an entire chapter to this issue.

1.5.4 How Queries and Updates Are Executed

Now that we have a rough picture of how data records are stored and efficiently accessed, the final issue that we want to consider is the execution of more complex queries and update operations. As an example consider a query that retrieves all persons of age less than 30 who live in Austin. In the standard query language SQL, this can be phrased as follows:

```
Select Name, City, Zipcode, Street
From Person
Where Age < 30 And City = "Austin"
```

The query decomposition and optimization layer translates this SQL query into an execution plan, taking into account the available indexes and other physical design features that were chosen by the database administrator. This translation involves solving an optimization problem, as its outcome should be the execution plan whose estimated execution cost is minimum or at least close to the minimum. Typically, query optimizers use a variety of sophisticated heuristics for this purpose. The resulting execution plan is usually represented as an operator tree. For our example, the operator tree of a possible execution plan is shown in Figure 1.7. This plan makes use of two indexes (the query optimizer must have determined that they both exist), namely, an index on the Age field and an index on the City field. It performs a range lookup on the Age index and an exact-match lookup on the City field. Both index lookups return RID lists that are then intersected to determine the RIDs of all qualifying records. Finally, the last step is to fetch those records based on their RIDs and perform some "output processing" such as projecting out the requested fields (i.e., dropping all other fields of the possibly long records).

An alternative execution plan is shown in Figure 1.8. In this plan, only one index is used, namely, the one on the City field. The reason for choosing this plan (over the previous one) could be that either the Age index does not exist, or is not very selective for the given range of key values, meaning that the RID list that would be returned from the range lookup is too large relative to the total number of records in the Person table.

The evaluation of such an operator tree proceeds bottom-up, from the leaves to the root. For each operator in the tree, the corresponding code in the database system fetches the operands and produces its output, which is then fed into the

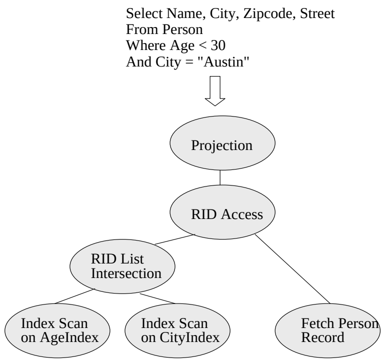

Figure 1.7: Example of a query execution plan.

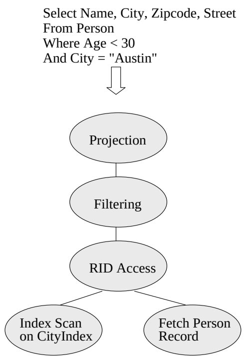

Figure 1.8: Alternative query execution plan for the query of Figure 1.7.

parent operator. As these operands as well as the intermediate outputs can become fairly large, the evaluation is usually carried out in a *pipelined* manner. So the output of an operator is not necessarily produced entirely before the parent operator is invoked. Rather, the parent operator is invoked as soon as its first input records are available. This evaluation principle establishes a set of producer-consumer data streams between operators, one stream for each operand arc in the tree. The benefit of the pipelining is that intermediate results do not have to be "materialized" in their entirety, and this is a significant cost saving, as the materialization would often require writing large intermediate output to disk and reading it again for the subsequently invoked parent operator. In a parallel database system, pipelining leads to true parallelism between operators, and, most importantly, it can be nicely combined with a parallelization of each operator (where the latter is the kind of parallelism that achieves the by far highest speedup).

1.6 Lessons Learned

This chapter has introduced transactions as a concept and emphasized their practical importance, from both an application and a system-architecture viewpoint. Transactions are essentially a "contract" between an application program and a collection of transactional servers that allows the program to combine a number of requests to these servers into a logical unit. The decisive part of this contract is that this unit has the ACID properties: atomicity, consistency-preservation, isolation, and durability. The benefit that we achieve from the transaction abstraction is twofold:

- Users (i.e., human "end-users") can rely on the consistency of the data that they see and maintain as part of their day-to-day business. This holds even for distributed, possibly heterogeneous information systems. Furthermore, this quality is not limited to database systems as the underlying data sources, but extends to document servers, mail servers, and other components of modern information systems.

- Application developers greatly benefit from the transaction concept in that the application programming is fundamentally simplified, thus boosting programming productivity and lowering the development and maintenance cost. Thanks to the ACID properties, application developers do not need to care about synchronizing concurrent or parallel data accesses or handling process and computer failures. Rather programs can be written as if each transaction had all underlying data exclusively for itself and none of the involved system components would ever fail.

These fundamental benefits are achieved by providing each data server, and possibly also other components of the information system, with two key pieces of software:

- a concurrency control component that guarantees the isolation of transactions in the presence of concurrent data accesses, and
- a recovery component that guarantees the atomicity and durability of transactions in the presence of failures.

This book is largely about models and algorithms for these two key building blocks of transactional information systems. The challenge lies in the performance of the concurrency control and recovery components: providing high concurrency, high throughput and fast response times, and ensuring fast recovery from failures and thus high availability with low overhead during normal operation.

Exercises

1. Consider an order-entry application, where customers order certain items from a retail seller. Customers place their orders over the phone, and agents enter these orders into a database at the merchant site. An order may consist of multiple order lines, each referring to one product with a specified quantity. Ordered products are shipped to the customer if they are available in sufficient quantity in the retail seller's warehouse. Customers pay by sending a check or through electronic funds transfer after receiving the shipped products.
   Discuss how ACID transactions can help maintaining the consistency of the data that underlies this kind of order-entry business.
   Hint: One possible approach, often used in practice, is to design and implement three basic transaction programs "NewOrder", "Shipment", and "Payment" for handling new orders, the shipping of ordered products, and the registration of payments. The underlying database would need the following tables with appropriate fields: Customers, Products, Orders, OrderLines.

2. Discuss further applications of database systems in the context of which the ACID properties of transactions make sense. Contrast these with other applications in which the ACID properties (totally or in part) appear too restrictive.

Bibliographical and Other Remarks

Various other textbooks have been published in recent years on the subject of transactions and transaction management; the most important ones of these are Bernstein et al. (1987), Papadimitriou (1986), Bernstein and Newcomer (1997), Cellary et al.

(1988), Lynch et al. (1994), and Gray and Reuter (1993). Additional books and original papers on specific subjects will be mentioned later.

Among the above textbooks, Bernstein and Newcomer (1997) as well as Gray and Reuter (1993) provide detailed considerations on the overall architecture and application-development aspects of transactional information systems devoting significant space to TP monitors and other types of request brokers. In this book, we rather focus on the fundamentals of transactional systems and do not further discuss the manifolded and continuously evolving issues of programming interfaces and commercial middleware technology. For the latter aspects refer, for example, to the recent books by Orfali et al. (1999) and Boucher and Katz (1999). We also will not go into more detail on the advanced application scenarios that we used in this chapter, specifically electronic commerce and workflow management. The interested reader is referred to the survey books by Adam et al. (1998) on e-commerce and Dogac et al. (1998), or Leymann and Roller (2000) on workflow technology. However, one should generally note that many aspects of these applications are still evolving so that the material in these books may become outdated fairly quickly.

Throughout the rest of this text we assume the reader is reasonably familiar with the fundamentals of database systems; we refer to Elmasri and Navathe (2000), Garcia et al. (2000), O'Neil (1994), Özsu and Valduriez (1999), Ramakrishnan and Gehrke (2000), Silberschatz et al. (1997), and Vossen (1991) for introductions to the topic in its full generality. Vossen (1992, 1996) offers surveys of the field which may be faster to accommodate. A good introduction to the specific issues of index structures and query processing in database systems is the book by Yu and Meng (1998). We will also refer to the reader's background on object-oriented programming and systems, in particular as used in the context of database systems; textbooks on this subject include Cattell (1994), Kemper and Moerkotte (1994), and Lausen and Vossen (1998).