C H A P T E R 2 4

Advanced Transaction Processing

In Chapters 15, 16, and 17, we introduced the concept of a transaction, which is a program unit that accesses—and possibly updates—various data items, and whose execution ensures the preservation of the ACID properties. We discussed in those chapters a variety of schemes for ensuring the ACID properties in an environment where failure can occur, and where the transactions may run concurrently.

In this chapter, we go beyond the basic schemes discussed previously, and cover advanced transaction-processing concepts, including transaction-processing monitors, transactional workflows, main-memory databases, real-time databases, long-duration transactions, nested transactions, and multidatabase transactions.

24.1 Transaction-Processing Monitors

Transaction-processing monitors (**TP monitors**) are systems that were developed in the 1970s and 1980s, initially in response to a need to support a large number of remote terminals (such as airline-reservation terminals) from a single computer. The term *TP monitor* initially stood for *teleprocessing monitor*.

TP monitors have since evolved to provide the core support for distributed transaction processing, and the term TP monitor has acquired its current meaning. The CICS TP monitor from IBM was one of the earliest TP monitors, and has been very widely used. Current-generation TP monitors include Tuxedo and Top End (both now from BEA Systems), Encina (from Transarc, which is now a part of IBM), and Transaction Server (from Microsoft).

24.1.1 TP-Monitor Architectures

Large-scale transaction processing systems are built around a client–server architecture. One way of building such systems is to have a server process for each client; the server performs authentication, and then executes actions requested by the client.

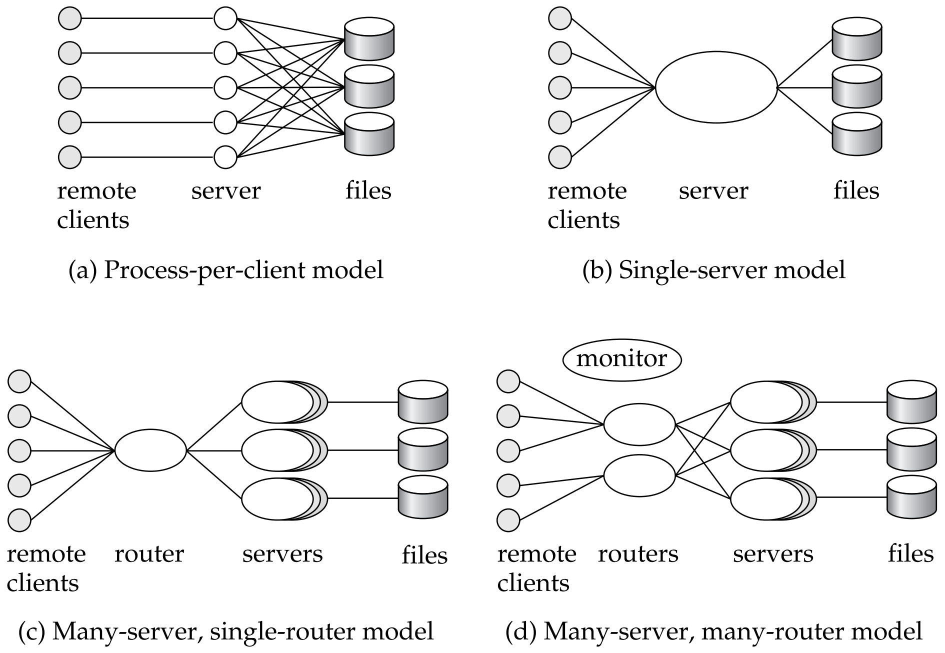

Figure 24.1 TP-monitor architectures.

This **process-per-client model** is illustrated in Figure 24.1a. This model presents several problems with respect to memory utilization and processing speed:

- Per-process memory requirements are high. Even if memory for program code is shared by all processes, each process consumes memory for local data and open file descriptors, as well as for operating-system overhead, such as page tables to support virtual memory.
- The operating system divides up available CPU time among processes by switching among them; this technique is called **multitasking**. Each **context switch** between one process and the next has considerable CPU overhead; even on today's fast systems, a context switch can take hundreds of microseconds.

The above problems can be avoided by having a single-server process to which all remote clients connect; this model is called the **single-server model**, illustrated in Figure 24.1b. Remote clients send requests to the server process, which then executes those requests. This model is also used in client–server environments, where clients send requests to a single-server process. The server process handles tasks, such as user authentication, that would normally be handled by the operating system. To avoid blocking other clients when processing a long request for one client, the server process is **multithreaded**: The server process has a thread of control for each client, and, in effect, implements its own low-overhead multitasking. It executes code on behalf of one client for a while, then saves the internal context and switches to the code for another client. Unlike the overhead of full multitasking, the cost of switching between threads is low (typically only a few microseconds).

Systems based on the single-server model, such as the original version of the IBM CICS TP monitor and file servers such as Novel's NetWare, successfully provided high transaction rates with limited resources. However, they had problems, especially when multiple applications accessed the same database:

- Since all the applications run as a single process, there is no protection among them. A bug in one application can affect all the other applications as well. It would be best to run each application as a separate process.
- Such systems are not suited for parallel or distributed databases, since a server process cannot execute on multiple computers at once. (However, concurrent threads within a process can be supported in a shared-memory multiprocessor system.) This is a serious drawback in large organizations, where parallel processing is critical for handling large workloads, and distributed data are becoming increasingly common.

One way to solve these problems is to run multiple application-server processes that access a common database, and to let the clients communicate with the application through a single communication process that routes requests. This model is called the **many-server, single-router model**, illustrated in Figure 24.1c. This model supports independent server processes for multiple applications; further, each application can have a pool of server processes, any one of which can handle a client session. The request can, for example, be routed to the most lightly loaded server in a pool. As before, each server process can itself be multithreaded, so that it can handle multiple clients concurrently. As a further generalization, the application servers can run on different sites of a parallel or distributed database, and the communication process can handle the coordination among the processes.

The above architecture is also widely used in Web servers. A Web server has a main process that receives HTTP requests, and then assigns the task of handling each request to a separate process (chosen from among a pool of processes). Each of the processes is itself multithreaded, so that it can handle multiple requests.

A more general architecture has multiple processes, rather than just one, to communicate with clients. The client communication processes interact with one or more router processes, which route their requests to the appropriate server. Later-generation TP monitors therefore have a different architecture, called the **many-server, many-router model**, illustrated in Figure 24.1d. A controller process starts up the other processes, and supervises their functioning. Tandem Pathway is an example of the later-generation TP monitors that use this architecture. Very high performance Web server systems also adopt such an architecture.

The detailed structure of a TP monitor appears in Figure 24.2. A TP monitor does more than simply pass messages to application servers. When messages arrive, they may have to be queued; thus, there is a **queue manager** for incoming messages. The queue may be a **durable queue**, whose entries survive system failures. Using a

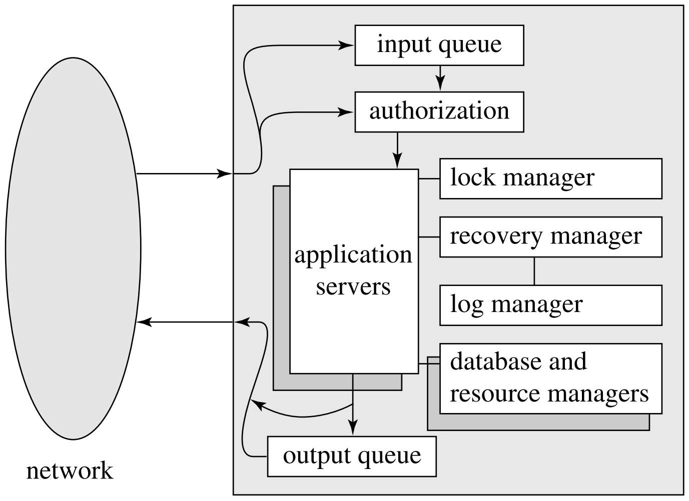

Figure 24.2 TP-monitor components.

durable queue helps ensure that once received and stored in the queue, the messages will be processed eventually, regardless of system failures. Authorization and application-server management (for example, server startup, and routing of messages to servers) are further functions of a TP monitor. TP monitors often provide logging, recovery, and concurrency-control facilities, allowing application servers to implement the ACID transaction properties directly if required.

Finally, TP monitors also provide support for persistent messaging. Recall that persistent messaging (Section 19.4.3) provides a guarantee that the message will be delivered if (and only if) the transaction commits.

In addition to these facilities, many TP monitors also provided *presentation facilities* to create menus/forms interfaces for dumb clients such as terminals; these facilities are no longer important since dumb clients are no longer widely used.

24.1.2 Application Coordination Using TP monitors

Applications today often have to interact with multiple databases. They may also have to interact with legacy systems, such as special-purpose data-storage systems built directly on file systems. Finally, they may have to communicate with users or other applications at remote sites. Hence, they also have to interact with communication subsystems. It is important to be able to coordinate data accesses, and to implement ACID properties for transactions across such systems.

Modern TP monitors provide support for the construction and administration of such large applications, built up from multiple subsystems such as databases, legacy systems, and communication systems. A TP monitor treats each subsystem as a **resource manager** that provides transactional access to some set of resources. The interface between the TP monitor and the resource manager is defined by a set of transaction primitives, such as *begin_transaction*, *commit_transaction*, *abort_transaction*, and *prepare_to_commit_transaction* (for two-phase commit). Of course, the resource manager must also provide other services, such as supplying data, to the application.

The resource-manager interface is defined by the X/Open Distributed Transaction Processing standard. Many database systems support the X/Open standards, and can act as resource managers. TP monitors—as well as other products, such as SQL systems, that support the X/Open standards—can connect to the resource managers.

In addition, services provided by a TP monitor, such as persistent messaging and durable queues, act as resource managers supporting transactions. The TP monitor can act as coordinator of two-phase commit for transactions that access these services as well as database systems. For example, when a queued update transaction is executed, an output message is delivered, and the request transaction is removed from the request queue. Two-phase commit between the database and the resource managers for the durable queue and persistent messaging helps ensure that, regardless of failures, either all these actions occur, or none occurs.

We can also use TP monitors to administer complex client–server systems consisting of multiple servers and a large number of clients. The TP monitor coordinates activities such as system checkpoints and shutdowns. It provides security and authentication of clients. It administers server pools by adding servers or removing servers without interruption of the system. Finally, it controls the scope of failures. If a server fails, the TP monitor can detect this failure, abort the transactions in progress, and restart the transactions. If a node fails, the TP monitor can migrate transactions to servers at other nodes, again backing out incomplete transactions. When failed nodes restart, the TP monitor can govern the recovery of the node's resource managers.

TP monitors can be used to hide database failures in replicated systems; remote backup systems (Section 17.10) are an example of replicated systems. Transaction requests are sent to the TP monitor, which relays the messages to one of the database replicas (the primary site, in case of remote backup systems). If one site fails, the TP monitor can transparently route messages to a backup site, masking the failure of the first site.

In client–server systems, clients often interact with servers via a **remote-procedure-call** (**RPC**) mechanism, where a client invokes a procedure call, which is actually executed at the server, with the results sent back to the client. As far as the client code that invokes the RPC is concerned, the call looks like a local procedure-call invocation. TP monitor systems, such as Encina, provide a **transactional RPC** interface to their services. In such an interface, the RPC mechanism provides calls that can be used to enclose a series of RPC calls within a transaction. Thus, updates performed by an RPC are carried out within the scope of the transaction, and can be rolled back if there is any failure.

24.2 Transactional Workflows

A **workflow** is an activity in which multiple tasks are executed in a coordinated way by different processing entities. A **task** defines some work to be done and can be specified in a number of ways, including a textual description in a file or electronic-mail message, a form, a message, or a computer program. The **processing entity** that

Workflow application	Typical task	Typical processing entity
electronic-mail routing	electronic-mail message	mailers
loan processing	form processing	humans, application software
purchase-order processing	form processing	humans, application software, DBMSs

Figure 24.3 Examples of workflows.

performs the tasks may be a person or a software system (for example, a mailer, an application program, or a database-management system).

Figure 24.3 shows examples of workflows. A simple example is that of an electronic-mail system. The delivery of a single mail message may involve several mailer systems that receive and forward the mail message, until the message reaches its destination, where it is stored. Each mailer performs a task—forwarding the mail to the next mailer—and the tasks of multiple mailers may be required to route mail from source to destination. Other terms used in the database and related literature to refer to workflows include **task flow** and **multisystem applications**. Workflow tasks are also sometimes called **steps**.

In general, workflows may involve one or more humans. For instance, consider the processing of a loan. The relevant workflow appears in Figure 24.4. The person who wants a loan fills out a form, which is then checked by a loan officer. An employee who processes loan applications verifies the data in the form, using sources such as credit-reference bureaus. When all the required information has been collected, the loan officer may decide to approve the loan; that decision may then have to be approved by one or more superior officers, after which the loan can be made. Each human here performs a task; in a bank that has not automated the task of loan processing, the coordination of the tasks is typically carried out by passing of the loan application, with attached notes and other information, from one employee to

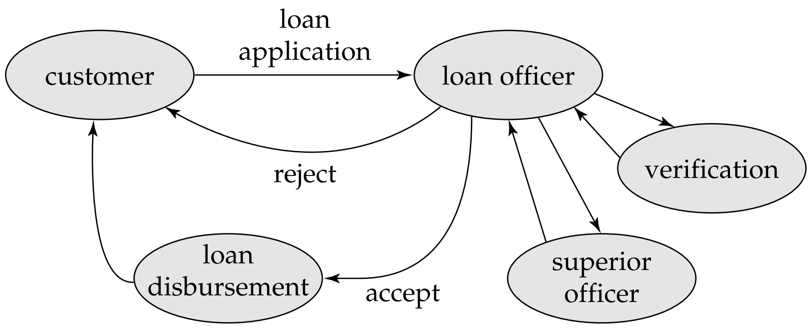

Figure 24.4 Workflow in loan processing.

the next. Other examples of workflows include processing of expense vouchers, of purchase orders, and of credit-card transactions.

Today, all the information related to a workflow is more than likely to be stored in a digital form on one or more computers, and, with the growth of networking, information can be easily transferred from one computer to another. Hence, it is feasible for organizations to automate their workflows. For example, to automate the tasks involved in loan processing, we can store the loan application and associated information in a database. The workflow itself then involves handing of responsibility from one human to the next, and possibly even to programs that can automatically fetch the required information. Humans can coordinate their activities by means such as electronic mail.

We have to address two activities, in general, to automate a workflow. The first is **workflow specification**: detailing the tasks that must be carried out and defining the execution requirements. The second problem is **workflow execution**, which we must do while providing the safeguards of traditional database systems related to computation correctness and data integrity and durability. For example, it is not acceptable for a loan application or a voucher to be lost, or to be processed more than once, because of a system crash. The idea behind transactional workflows is to use and extend the concepts of transactions to the context of workflows.

Both activities are complicated by the fact that many organizations use several independently managed information-processing systems that, in most cases, were developed separately to automate different functions. Workflow activities may require interactions among several such systems, each performing a task, as well as interactions with humans.

A number of workflow systems have been developed in recent years. Here, we study properties of workflow systems at a relatively abstract level, without going into the details of any particular system.

24.2.1 Workflow Specification

Internal aspects of a task do not need to be modeled for the purpose of specification and management of a workflow. In an abstract view of a task, a task may use parameters stored in its input variables, may retrieve and update data in the local system, may store its results in its output variables, and may be queried about its execution state. At any time during the execution, the **workflow state** consists of the collection of states of the workflow’s constituent tasks, and the states (values) of all variables in the workflow specification.

The coordination of tasks can be specified either statically or dynamically. A static specification defines the tasks—and dependencies among them—before the execution of the workflow begins. For instance, the tasks in an expense-voucher workflow may consist of the approvals of the voucher by a secretary, a manager, and an accountant, in that order, and finally by the delivery of a check. The dependencies among the tasks may be simple—each task has to be completed before the next begins.

A generalization of this strategy is to have a precondition for execution of each task in the workflow, so that all possible tasks in a workflow and their dependencies are known in advance, but only those tasks whose preconditions are satisfied are executed. The preconditions can be defined through dependencies such as the following:

- **Execution states** of other tasks—for example, "task t_i cannot start until task t_j has ended," or "task t_i must abort if task t_j has committed"
- **Output values** of other tasks—for example, "task t_i can start if task t_j returns a value greater than 25," or "the manager-approval task can start if the secretary-approval task returns a value of OK"
- **External variables** modified by external events—for example, "task t_i cannot be started before 9 AM," or "task t_i must be started within 24 hours of the completion of task t_j"

We can combine the dependencies by the regular logical connectors (**or**, **and**, **not**) to form complex scheduling preconditions.

An example of dynamic scheduling of tasks is an electronic-mail routing system. The next task to be scheduled for a given mail message depends on what the destination address of the message is, and on which intermediate routers are functioning.

24.2.2 Failure-Atomicity Requirements of a Workflow

The workflow designer may specify the **failure-atomicity** requirements of a workflow according to the semantics of the workflow. The traditional notion of failure atomicity would require that a failure of any task results in the failure of the workflow. However, a workflow can, in many cases, survive the failure of one of its tasks—for example, by executing a functionally equivalent task at another site. Therefore, we should allow the designer to define failure-atomicity requirements of a workflow. The system must guarantee that every execution of a workflow will terminate in a state that satisfies the failure-atomicity requirements defined by the designer. We call those states **acceptable termination states** of a workflow. All other execution states of a workflow constitute a set of **nonacceptable termination states**, in which the failure-atomicity requirements may be violated.

An acceptable termination state can be designated as committed or aborted. A **committed acceptable termination state** is an execution state in which the objectives of a workflow have been achieved. In contrast, an **aborted acceptable termination state** is a valid termination state in which a workflow has failed to achieve its objectives. If an aborted acceptable termination state has been reached, all undesirable effects of the partial execution of the workflow must be undone in accordance with that workflow's failure-atomicity requirements.

A workflow must reach an acceptable termination state *even in the presence of system failures*. Thus, if a workflow was in a nonacceptable termination state at the time of failure, during system recovery it must be brought to an acceptable termination state (whether aborted or committed).

For example, in the loan-processing workflow, in the final state, either the loan applicant is told that a loan cannot be made or the loan is disbursed. In case of failures such as a long failure of the verification system, the loan application could be returned to the loan applicant with a suitable explanation; this outcome would constitute an aborted acceptable termination. A committed acceptable termination would be either the acceptance or the rejection of the loan.

In general, a task can commit and release its resources before the workflow reaches a termination state. However, if the multitask transaction later aborts, its failure atomicity may require that we undo the effects of already completed tasks (for example, committed subtransactions) by executing compensating tasks (as subtransactions). The semantics of compensation requires that a compensating transaction eventually complete its execution successfully, possibly after a number of resubmissions.

In an expense-voucher-processing workflow, for example, a department-budget balance may be reduced on the basis of an initial approval of a voucher by the manager. If the voucher is later rejected, whether because of failure or for other reasons, the budget may have to be restored by a compensating transaction.

24.2.3 Execution of Workflows

The execution of the tasks may be controlled by a human coordinator or by a software system called a **workflow-management system**. A workflow-management system consists of a scheduler, task agents, and a mechanism to query the state of the workflow system. A task agent controls the execution of a task by a processing entity. A scheduler is a program that processes workflows by submitting various tasks for execution, monitoring various events, and evaluating conditions related to intertask dependencies. A scheduler may submit a task for execution (to a task agent), or may request that a previously submitted task be aborted. In the case of multidatabase transactions, the tasks are subtransactions, and the processing entities are local database management systems. In accordance with the workflow specifications, the scheduler enforces the scheduling dependencies and is responsible for ensuring that tasks reach acceptable termination states.

There are three architectural approaches to the development of a workflow-management system. A **centralized architecture** has a single scheduler that schedules the tasks for all concurrently executing workflows. The **partially distributed architecture** has one scheduler instantiated for each workflow. When the issues of concurrent execution can be separated from the scheduling function, the latter option is a natural choice. A **fully distributed architecture** has no scheduler, but the task agents coordinate their execution by communicating with one another to satisfy task dependencies and other workflow execution requirements.

The simplest workflow-execution systems follow the fully distributed approach just described and are based on messaging. Messaging may be implemented by persistent messaging mechanisms, to provide guaranteed delivery. Some implementations use e-mail for messaging; such implementations provide many of the features of persistent messaging, but generally do not guarantee atomicity of message delivery and transaction commit. Each site has a task agent that executes tasks received through messages. Execution may also involve presenting messages to humans, who have then to carry out some action. When a task is completed at a site, and needs to be processed at another site, the task agent dispatches a message to the next site. The message contains all relevant information about the task to be performed. Such message-based workflow systems are particularly useful in networks that may be disconnected for part of the time, such as dial-up networks.

The centralized approach is used in workflow systems where the data are stored in a central database. The scheduler notifies various agents, such as humans or computer programs, that a task has to be carried out, and keeps track of task completion. It is easier to keep track of the state of a workflow with a centralized approach than it is with a fully distributed approach.

The scheduler must guarantee that a workflow will terminate in one of the specified acceptable termination states. Ideally, before attempting to execute a workflow, the scheduler should examine that workflow to check whether the workflow may terminate in a nonacceptable state. If the scheduler cannot guarantee that a workflow will terminate in an acceptable state, it should reject such specifications without attempting to execute the workflow. As an example, let us consider a workflow consisting of two tasks represented by subtransactions S_1 and S_2, with the failure-atomicity requirements indicating that either both or neither of the subtransactions should be committed. If S_1 and S_2 do not provide prepared-to-commit states (for a two-phase commit), and further do not have compensating transactions, then it is possible to reach a state where one subtransaction is committed and the other aborted, and there is no way to bring both to the same state. Therefore, such a workflow specification is **unsafe**, and should be rejected.

Safety checks such as the one just described may be impossible or impractical to implement in the scheduler; it then becomes the responsibility of the person designing the workflow specification to ensure that the workflows are safe.

24.2.4 Recovery of a Workflow

The objective of **workflow recovery** is to enforce the failure atomicity of the workflows. The recovery procedures must make sure that, if a failure occurs in any of the workflow-processing components (including the scheduler), the workflow will eventually reach an acceptable termination state (whether aborted or committed). For example, the scheduler could continue processing after failure and recovery, as though nothing happened, thus providing forward recoverability. Otherwise, the scheduler could abort the whole workflow (that is, reach one of the global abort states). In either case, some subtransactions may need to be committed or even submitted for execution (for example, compensating subtransactions).

We assume that the processing entities involved in the workflow have their own local recovery systems and handle their local failures. To recover the execution-environment context, the failure-recovery routines need to restore the state information of the scheduler at the time of failure, including the information about the execution states of each task. Therefore, the appropriate status information must be logged on stable storage.

We also need to consider the contents of the message queues. When one agent hands off a task to another, the handoff should be carried out exactly once: If the handoff happens twice a task may get executed twice; if the handoff does not occur, the task may get lost. Persistent messaging (Section 19.4.3) provides exactly the features to ensure positive, single handoff.

24.2.5 Workflow Management Systems

Workflows are often hand coded as part of application systems. For instance, enterprise resource planning (ERP) systems, which help coordinate activities across an entire enterprise, have numerous workflows built into them.

The goal of workflow management systems is to simplify the construction of workflows and make them more reliable, by permitting them to be specified in a high-level manner and executed in accordance with the specification. There are a large number of commercial workflow management systems; some, like FlowMark from IBM, are general-purpose workflow management systems, while others are specific to particular workflows, such as order processing or bug/failure reporting systems.

In today's world of interconnected organizations, it is not sufficient to manage workflows only within an organization. Workflows that cross organizational boundaries are becoming increasingly common. For instance, consider an order placed by an organization and communicated to another organization that fulfills the order. In each organization there may be a workflow associated with the order, and it is important that the workflows be able to interoperate, in order to minimize human intervention.

The Workflow Management Coalition has developed standards for interoperation between workflow systems. Current standardization efforts use XML as the underlying language for communicating information about the workflow. See the bibliographical notes for more information.

24.3 Main-Memory Databases

To allow a high rate of transaction processing (hundreds or thousands of transactions per second), we must use high-performance hardware, and must exploit parallelism. These techniques alone, however, are insufficient to obtain very low response times, since disk I/O remains a bottleneck—about 10 milliseconds are required for each I/O and this number has not decreased at a rate comparable to the increase in processor speeds. Disk I/O is often the bottleneck for reads, as well as for transaction commits. The long disk latency (about 10 milliseconds average) increases not only the time to access a data item, but also limits the number of accesses per second.

We can make a database system less disk bound by increasing the size of the database buffer. Advances in main-memory technology let us construct large main memories at relatively low cost. Today, commercial 64-bit systems can support main memories of tens of gigabytes.

For some applications, such as real-time control, it is necessary to store data in main memory to meet performance requirements. The memory size required for most such systems is not exceptionally large, although there are at least a few applications that require multiple gigabytes of data to be memory resident. Since memory sizes have been growing at a very fast rate, an increasing number of applications can be expected to have data that fit into main memory.

Large main memories allow faster processing of transactions, since data are memory resident. However, there are still disk-related limitations:

- Log records must be written to stable storage before a transaction is committed. The improved performance made possible by a large main memory may result in the logging process becoming a bottleneck. We can reduce commit time by creating a stable log buffer in main memory, using nonvolatile RAM (implemented, for example, by battery backed-up memory). The overhead imposed by logging can also be reduced by the *group-commit* technique discussed later in this section. Throughput (number of transactions per second) is still limited by the data-transfer rate of the log disk.
- Buffer blocks marked as modified by committed transactions still have to be written so that the amount of log that has to be replayed at recovery time is reduced. If the update rate is extremely high, the disk data-transfer rate may become a bottleneck.
- If the system crashes, all of main memory is lost. On recovery, the system has an empty database buffer, and data items must be input from disk when they are accessed. Therefore, even after recovery is complete, it takes some time before the database is fully loaded in main memory and high-speed processing of transactions can resume.

On the other hand, a main-memory database provides opportunities for optimizations:

- Since memory is costlier than disk space, internal data structures in main-memory databases have to be designed to reduce space requirements. However, data structures can have pointers crossing multiple pages unlike those in disk databases, where the cost of the I/Os to traverse multiple pages would be excessively high. For example, tree structures in main-memory databases can be relatively deep, unlike B$^+$-trees, but should minimize space requirements.
- There is no need to pin buffer pages in memory before data are accessed, since buffer pages will never be replaced.
- Query-processing techniques should be designed to minimize space overhead, so that main memory limits are not exceeded while a query is being evaluated; that situation would result in paging to swap area, and would slow down query processing.
- Once the disk I/O bottleneck is removed, operations such as locking and latching may become bottlenecks. Such bottlenecks must be eliminated by improvements in the implementation of these operations.
- Recovery algorithms can be optimized, since pages rarely need to be written out to make space for other pages.

TimesTen and DataBlitz are two main-memory database products that exploit several of these optimizations, while the Oracle database has added special features to support very large main memories. Additional information on main-memory databases is given in the references in the bibliographical notes.

The process of committing a transaction *T* requires these records to be written to stable storage:

- All log records associated with *T* that have not been output to stable storage
- The <T **commit**> log record

These output operations frequently require the output of blocks that are only partially filled. To ensure that nearly full blocks are output, we use the **group-commit** technique. Instead of attempting to commit *T* when *T* completes, the system waits until several transactions have completed, or a certain period of time has passed since a transaction completed execution. It then commits the group of transactions that are waiting, together. Blocks written to the log on stable storage would contain records of several transactions. By careful choice of group size and maximum waiting time, the system can ensure that blocks are full when they are written to stable storage without making transactions wait excessively. This technique results, on average, in fewer output operations per committed transaction.

Although group commit reduces the overhead imposed by logging, it results in a slight delay in commit of transactions that perform updates. The delay can be made quite small (say, 10 milliseconds), which is acceptable for many applications. These delays can be eliminated if disks or disk controllers support nonvolatile RAM buffers for write operations. Transactions can commit as soon as the write is performed on the nonvolatile RAM buffer. In this case, there is no need for group commit.

Note that group commit is useful even in databases with disk-resident data.

24.4 Real-Time Transaction Systems

The integrity constraints that we have considered thus far pertain to the values stored in the database. In certain applications, the constraints include **deadlines** by which a task must be completed. Examples of such applications include plant management, traffic control, and scheduling. When deadlines are included, correctness of an execution is no longer solely an issue of database consistency. Rather, we are concerned with how many deadlines are missed, and by how much time they are missed. Deadlines are characterized as follows:

- **Hard deadline**. Serious problems, such as system crash, may occur if a task is not completed by its deadline.
- **Firm deadline**. The task has zero value if it is completed after the deadline.
- **Soft deadlines**. The task has diminishing value if it is completed after the deadline, with the value approaching zero as the degree of lateness increases.

Systems with deadlines are called **real-time systems**.

Transaction management in real-time systems must take deadlines into account. If the concurrency-control protocol determines that a transaction T_i must wait, it may cause T_i to miss the deadline. In such cases, it may be preferable to pre-empt the transaction holding the lock, and to allow T_i to proceed. Pre-emption must be used with care, however, because the time lost by the pre-empted transaction (due to rollback and restart) may cause the transaction to miss its deadline. Unfortunately, it is difficult to determine whether rollback or waiting is preferable in a given situation.

A major difficulty in supporting real-time constraints arises from the variance in transaction execution time. In the best case, all data accesses reference data in the database buffer. In the worst case, each access causes a buffer page to be written to disk (preceded by the requisite log records), followed by the reading from disk of the page containing the data to be accessed. Because the two or more disk accesses required in the worst case take several orders of magnitude more time than the main-memory references required in the best case, transaction execution time can be estimated only very poorly if data are resident on disk. Hence, main-memory databases are often used if real-time constraints have to be met.

However, even if data are resident in main memory, variances in execution time arise from lock waits, transaction aborts, and so on. Researchers have devoted considerable effort to concurrency control for real-time databases. They have extended locking protocols to provide higher priority for transactions with early deadlines. They have found that optimistic concurrency protocols perform well in real-time databases; that is, these protocols result in fewer missed deadlines than even the extended locking protocols. The bibliographical notes provide references to research in the area of real-time databases.

In real-time systems, deadlines, rather than absolute speed, are the most important issue. Designing a real-time system involves ensuring that there is enough processing power to meet deadlines without requiring excessive hardware resources. Achieving this objective, despite the variance in execution time resulting from transaction management, remains a challenging problem.

24.5 Long-Duration Transactions

The transaction concept developed initially in the context of data-processing applications, in which most transactions are noninteractive and of short duration. Although the techniques presented here and earlier in Chapters 15, 16, and 17 work well in those applications, serious problems arise when this concept is applied to database systems that involve human interaction. Such transactions have these key properties:

- **Long duration**. Once a human interacts with an active transaction, that transaction becomes a **long-duration transaction** from the perspective of the computer, since human response time is slow relative to computer speed. Furthermore, in design applications, the human activity may involve hours, days, or an even longer period. Thus, transactions may be of long duration in human terms, as well as in machine terms.

- **Exposure of uncommitted data**. Data generated and displayed to a user by a long-duration transaction are uncommitted, since the transaction may abort. Thus, users—and, as a result, other transactions—may be forced to read uncommitted data. If several users are cooperating on a project, user transactions may need to exchange data prior to transaction commit.

- **Subtasks**. An interactive transaction may consist of a set of subtasks initiated by the user. The user may wish to abort a subtask without necessarily causing the entire transaction to abort.
- **Recoverability**. It is unacceptable to abort a long-duration interactive transaction because of a system crash. The active transaction must be recovered to a state that existed shortly before the crash so that relatively little human work is lost.
- **Performance**. Good performance in an interactive transaction system is defined as fast response time. This definition is in contrast to that in a noninteractive system, in which high throughput (number of transactions per second) is the goal. Systems with high throughput make efficient use of system resources. However, in the case of interactive transactions, the most costly resource is the user. If the efficiency and satisfaction of the user is to be optimized, response time should be fast (from a human perspective). In those cases where a task takes a long time, response time should be predictable (that is, the variance in response times should be low), so that users can manage their time well.

In Sections 24.5.1 through 24.5.5, we shall see why these five properties are incompatible with the techniques presented thus far, and shall discuss how those techniques can be modified to accommodate long-duration interactive transactions.

24.5.1 Nonserializable Executions

The properties that we discussed make it impractical to enforce the requirement used in earlier chapters that only serializable schedules be permitted. Each of the concurrency-control protocols of Chapter 16 has adverse effects on long-duration transactions:

- **Two-phase locking**. When a lock cannot be granted, the transaction requesting the lock is forced to wait for the data item in question to be unlocked. The duration of this wait is proportional to the duration of the transaction holding the lock. If the data item is locked by a short-duration transaction, we expect that the waiting time will be short (except in case of deadlock or extraordinary system load). However, if the data item is locked by a long-duration transaction, the wait will be of long duration. Long waiting times lead to both longer response time and an increased chance of deadlock.
- **Graph-based protocols**. Graph-based protocols allow for locks to be released earlier than under the two-phase locking protocols, and they prevent deadlock. However, they impose an ordering on the data items. Transactions must lock data items in a manner consistent with this ordering. As a result, a transaction may have to lock more data than it needs. Furthermore, a transaction must hold a lock until there is no chance that the lock will be needed again. Thus, long-duration lock waits are likely to occur.

- **Timestamp-based protocols**. Timestamp protocols never require a transaction to wait. However, they do require transactions to abort under certain circumstances. If a long-duration transaction is aborted, a substantial amount of work is lost. For noninteractive transactions, this lost work is a performance issue. For interactive transactions, the issue is also one of user satisfaction. It is highly undesirable for a user to find that several hours' worth of work have been undone.
- **Validation protocols**. Like timestamp-based protocols, validation protocols enforce serializability by means of transaction abort.

Thus, it appears that the enforcement of serializability results in long-duration waits, in abort of long-duration transactions, or in both. There are theoretical results, cited in the bibliographical notes, that substantiate this conclusion.

Further difficulties with the enforcement of serializability arise when we consider recovery issues. We previously discussed the problem of cascading rollback, in which the abort of a transaction may lead to the abort of other transactions. This phenomenon is undesirable, particularly for long-duration transactions. If locking is used, exclusive locks must be held until the end of the transaction, if cascading rollback is to be avoided. This holding of exclusive locks, however, increases the length of transaction waiting time.

Thus, it appears that the enforcement of transaction atomicity must either lead to an increased probability of long-duration waits or create a possibility of cascading rollback.

These considerations are the basis for the alternative concepts of correctness of concurrent executions and transaction recovery that we consider in the remainder of this section.

24.5.2 Concurrency Control

The fundamental goal of database concurrency control is to ensure that concurrent execution of transactions does not result in a loss of database consistency. The concept of serializability can be used to achieve this goal, since all serializable schedules preserve consistency of the database. However, not all schedules that preserve consistency of the database are serializable. For an example, consider again a bank database consisting of two accounts A and B, with the consistency requirement that the sum $A + B$ be preserved. Although the schedule of Figure 24.5 is not conflict serializable, it nevertheless preserves the sum of $A + B$. It also illustrates two important points about the concept of correctness without serializability.

- Correctness depends on the specific consistency constraints for the database.
- Correctness depends on the properties of operations performed by each transaction.

In general it is not possible to perform an automatic analysis of low-level operations by transactions and check their effect on database consistency constraints. However, there are simpler techniques. One is to use the database consistency constraints as

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B - 10$
	write(B)
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$A := A + 10$
	write(A)

Figure 24.5 A non-conflict-serializable schedule.

the basis for a split of the database into subdatabases on which concurrency can be managed separately. Another is to treat some operations besides **read** and **write** as fundamental low-level operations, and to extend concurrency control to deal with them.

The bibliographical notes reference other techniques for ensuring consistency without requiring serializability. Many of these techniques exploit variants of multiversion concurrency control (see Section 17.6). For older data-processing applications that need only one version, multiversion protocols impose a high space overhead to store the extra versions. Since many of the new database applications require the maintenance of versions of data, concurrency-control techniques that exploit multiple versions are practical.

24.5.3 Nested and Multilevel Transactions

A long-duration transaction can be viewed as a collection of related subtasks or subtransactions. By structuring a transaction as a set of subtransactions, we are able to enhance parallelism, since it may be possible to run several subtransactions in parallel. Furthermore, it is possible to deal with failure of a subtransaction (due to abort, system crash, and so on) without having to roll back the entire long-duration transaction.

A nested or multilevel transaction T consists of a set $T = \{t_1, t_2, \ldots, t_n\}$ of subtransactions and a partial order P on T. A subtransaction t_i in T may abort without forcing T to abort. Instead, T may either restart t_i or simply choose not to run t_i. If t_i commits, this action does not make t_i permanent (unlike the situation in Chapter 17). Instead, t_i *commits to* T, and may still abort (or require compensation—see Section 24.5.4) if T aborts. An execution of T must not violate the partial order P. That is, if an edge $t_i \rightarrow t_j$ appears in the precedence graph, then $t_j \rightarrow t_i$ must not be in the transitive closure of P.

Nesting may be several levels deep, representing a subdivision of a transaction into subtasks, subsubtasks, and so on. At the lowest level of nesting, we have the standard database operations **read** and **write** that we have used previously.

If a subtransaction of T is permitted to release locks on completion, T is called a **multilevel transaction**. When a multilevel transaction represents a long-duration activity, the transaction is sometimes referred to as a **saga**. Alternatively, if locks held by a subtransaction t_i of T are automatically assigned to T on completion of t_i, T is called a **nested transaction**.

Although the main practical value of multilevel transactions arises in complex, long-duration transactions, we shall use the simple example of Figure 24.5 to show how nesting can create higher-level operations that may enhance concurrency. We rewrite transaction T_1, using subtransactions $T_{1,1}$ and $T_{1,2}$, which perform increment or decrement operations:

- T_1 consists of
  - ☐ $T_{1,1}$, which subtracts 50 from A
  - ☐ $T_{1,2}$, which adds 50 to B

Similarly, we rewrite transaction T_2, using subtransactions $T_{2,1}$ and $T_{2,2}$, which also perform increment or decrement operations:

- T_2 consists of
  - ☐ $T_{2,1}$, which subtracts 10 from B
  - ☐ $T_{2,2}$, which adds 10 to A

No ordering is specified on $T_{1,1},\ T_{1,2},\ T_{2,1}$, and $T_{2,2}$. Any execution of these subtransactions will generate a correct result. The schedule of Figure 24.5 corresponds to the schedule $< T_{1,1},\ T_{2,1},\ T_{1,2},\ T_{2,2} >$.

24.5.4 Compensating Transactions

To reduce the frequency of long-duration waiting, we arrange for uncommitted updates to be exposed to other concurrently executing transactions. Indeed, multilevel transactions may allow this exposure. However, the exposure of uncommitted data creates the potential for cascading rollbacks. The concept of **compensating transactions** helps us to deal with this problem.

Let transaction T be divided into several subtransactions $t_1, t_2, \ldots, t_n$. After a subtransaction t_i commits, it releases its locks. Now, if the outer-level transaction T has to be aborted, the effect of its subtransactions must be undone. Suppose that subtransactions $t_1, \ldots, t_k$ have committed, and that t_{k+1} was executing when the decision to abort is made. We can undo the effects of t_{k+1} by aborting that subtransaction. However, it is not possible to abort subtransactions $t_1, \ldots, t_k$, since they have committed already.

Instead, we execute a new subtransaction ct_i, called a *compensating transaction*, to undo the effect of a subtransaction t_i. Each subtransaction t_i is required to have a compensating transaction ct_i. The compensating transactions must be executed in the inverse order $ct_k, \ldots, ct_1$. Here are several examples of compensation:

- Consider the schedule of Figure 24.5, which we have shown to be correct, although not conflict serializable. Each subtransaction releases its locks once it completes. Suppose that T_2 fails just prior to termination, after $T_{2,2}$ has released its locks. We then run a compensating transaction for $T_{2,2}$ that subtracts 10 from A and a compensating transaction for $T_{2,1}$ that adds 10 to B.
- Consider a database insert by transaction T_i that, as a side effect, causes a B$^+$-tree index to be updated. The insert operation may have modified several nodes of the B$^+$-tree index. Other transactions may have read these nodes in accessing data other than the record inserted by T_i. As in Section 17.9, we can undo the insertion by deleting the record inserted by T_i. The result is a correct, consistent B$^+$-tree, but is not necessarily one with exactly the same structure as the one we had before T_i started. Thus, deletion is a compensating action for insertion.
- Consider a long-duration transaction T_i representing a travel reservation. Transaction T has three subtransactions: $T_{i,1}$, which makes airline reservations; $T_{i,2}$, which reserves rental cars; and $T_{i,3}$, which reserves a hotel room. Suppose that the hotel cancels the reservation. Instead of undoing all of T_i, we compensate for the failure of $T_{i,3}$ by deleting the old hotel reservation and making a new one.

If the system crashes in the middle of executing an outer-level transaction, its subtransactions must be rolled back when it recovers. The techniques described in Section 17.9 can be used for this purpose.

Compensation for the failure of a transaction requires that the semantics of the failed transaction be used. For certain operations, such as incrementation or insertion into a B$^+$-tree, the corresponding compensation is easily defined. For more complex transactions, the application programmers may have to define the correct form of compensation at the time that the transaction is coded. For complex interactive transactions, it may be necessary for the system to interact with the user to determine the proper form of compensation.

24.5.5 Implementation Issues

The transaction concepts discussed in this section create serious difficulties for implementation. We present a few of them here, and discuss how we can address these problems.

Long-duration transactions must survive system crashes. We can ensure that they will by performing a **redo** on committed subtransactions, and by performing either an **undo** or compensation for any short-duration subtransactions that were active at the time of the crash. However, these actions solve only part of the problem. In typical database systems, such internal system data as lock tables and transactions timestamps are kept in volatile storage. For a long-duration transaction to be resumed after a crash, these data must be restored. Therefore, it is necessary to log not only changes to the database, but also changes to internal system data pertaining to long-duration transactions.

Logging of updates is made more complex when certain types of data items exist in the database. A data item may be a CAD design, text of a document, or another form of composite design. Such data items are physically large. Thus, storing both the old and new values of the data item in a log record is undesirable.

There are two approaches to reducing the overhead of ensuring the recoverability of large data items:

- **Operation logging**. Only the operation performed on the data item and the data-item name are stored in the log. Operation logging is also called **logical logging**. For each operation, an inverse operation must exist. We perform **undo** using the inverse operation, and **redo** using the operation itself. Recovery through operation logging is more difficult, since **redo** and **undo** are not idempotent. Further, using logical logging for an operation that updates multiple pages is greatly complicated by the fact that some, but not all, of the updated pages may have been written to the disk, so it is hard to apply either the **redo** or the **undo** of the operation on the disk image during recovery.

  Using physical redo logging and logical undo logging, as described in Section 17.9, provides the concurrency benefits of logical logging while avoiding the above pitfalls.

- **Logging and shadow paging**. Logging is used for modifications to small data items, but large data items are made recoverable via a shadow-page technique (see Section 17.5). When we use shadowing, only those pages that are actually modified need to be stored in duplicate.

Regardless of the technique used, the complexities introduced by long-duration transactions and large data items complicate the recovery process. Thus, it is desirable to allow certain noncritical data to be exempt from logging, and to rely instead on offline backups and human intervention.

24.6 Transaction Management in Multidatabases

Recall from Section 19.8 that a multidatabase system creates the illusion of logical database integration, in a heterogeneous database system where the local database systems may employ different logical data models and data-definition and data-manipulation languages, and may differ in their concurrency-control and transaction-management mechanisms.

A multidatabase system supports two types of transactions:

1. **Local transactions**. These transactions are executed by each local database system outside of the multidatabase system's control.
2. **Global transactions**. These transactions are executed under the multidatabase system's control.

The multidatabase system is aware of the fact that local transactions may run at the local sites, but it is not aware of what specific transactions are being executed, or of what data they may access.

Ensuring the local autonomy of each database system requires that no changes be made to its software. A database system at one site thus is not able to communicate directly with a one at any other site to synchronize the execution of a global transaction active at several sites.

Since the multidatabase system has no control over the execution of local transactions, each local system must use a concurrency-control scheme (for example, two-phase locking or timestamping) to ensure that its schedule is serializable. In addition, in case of locking, the local system must be able to guard against the possibility of local deadlocks.

The guarantee of local serializability is not sufficient to ensure global serializability. As an illustration, consider two global transactions T_1 and T_2, each of which accesses and updates two data items, A and B, located at sites S_1 and S_2, respectively. Suppose that the local schedules are serializable. It is still possible to have a situation where, at site S_1, T_2 follows T_1, whereas, at S_2, T_1 follows T_2, resulting in a nonserializable global schedule. Indeed, even if there is no concurrency among global transactions (that is, a global transaction is submitted only after the previous one commits or aborts), local serializability is not sufficient to ensure global serializability (see Exercise 24.14).

Depending on the implementation of the local database systems, a global transaction may not be able to control the precise locking behavior of its local substransactions. Thus, even if all local database systems follow two-phase locking, it may be possible only to ensure that each local transaction follows the rules of the protocol. For example, one local database system may commit its subtransaction and release locks, while the subtransaction at another local system is still executing. If the local systems permit control of locking behavior and all systems follow two-phase locking, then the multidatabase system can ensure that global transactions lock in a two-phase manner and the lock points of conflicting transactions would then define their global serialization order. If different local systems follow different concurrency-control mechanisms, however, this straightforward sort of global control does not work.

There are many protocols for ensuring consistency despite concurrent execution of global and local transactions in multidatabase systems. Some are based on imposing sufficient conditions to ensure global serializability. Others ensure only a form of consistency weaker than serializability, but achieve this consistency by less restrictive means. We consider one of the latter schemes: *two-level serializability*. Section 24.5 describes further approaches to consistency without serializability; other approaches are cited in the bibliographical notes.

A related problem in multidatabase systems is that of global atomic commit. If all local systems follow the two-phase commit protocol, that protocol can be used to achieve global atomicity. However, local systems not designed to be part of a distributed system may not be able to participate in such a protocol. Even if a local system is capable of supporting two-phase commit, the organization owning the system may be unwilling to permit waiting in cases where blocking occurs. In such cases, compromises may be made that allow for lack of atomicity in certain failure modes. Further discussion of these matters appears in the literature (see the bibliographical notes).

24.6.1 Two-Level Serializability

Two-level serializability (**2LSR**) ensures serializability at two levels of the system:

- Each local database system ensures local serializability among its local transactions, including those that are part of a global transaction.
- The multidatabase system ensures serializability among the global transactions alone—*ignoring the orderings induced by local transactions*.

Each of these serializability levels is simple to enforce. Local systems already offer guarantees of serializability; thus, the first requirement is easy to achieve. The second requirement applies to only a projection of the global schedule in which local transactions do not appear. Thus, the multidatabase system can ensure the second requirement using standard concurrency-control techniques (the precise choice of technique does not matter).

The two requirements of 2LSR are not sufficient to ensure global serializability. However, under the 2LSR-based approach, we adopt a requirement weaker than serializability, called **strong correctness**:

1. Preservation of consistency as specified by a set of consistency constraints
2. Guarantee that the set of data items read by each transaction is consistent

It can be shown that certain restrictions on transaction behavior, combined with 2LSR, are sufficient to ensure strong correctness (although not necessarily to ensure serializability). We list several of these restrictions.

In each of the protocols, we distinguish between **local data** and **global data**. Local data items belong to a particular site and are under the sole control of that site. Note that there cannot be any consistency constraints between local data items at distinct sites. Global data items belong to the multidatabase system, and, though they may be stored at a local site, are under the control of the multidatabase system.

The **global-read protocol** allows global transactions to read, but not to update, local data items, while disallowing all access to global data by local transactions. The global-read protocol ensures strong correctness if all these conditions hold:

1. Local transactions access only local data items.
2. Global transactions may access global data items, and may read local data items (although they must not write local data items).
3. There are no consistency constraints between local and global data items.

The **local-read protocol** grants local transactions read access to global data, but disallows all access to local data by global transactions. In this protocol, we need to introduce the notion of a **value dependency**. A transaction has a value dependency if the value that it writes to a data item at one site depends on a value that it read for a data item on another site.

The local-read protocol ensures strong correctness if all these conditions hold:

1. Local transactions may access local data items, and may read global data items stored at the site (although they must not write global data items).
2. Global transactions access only global data items.
3. No transaction may have a value dependency.

The **global-read–write/local-read protocol** is the most generous in terms of data access of the protocols that we have considered. It allows global transactions to read and write local data, and allows local transactions to read global data. However, it imposes both the value-dependency condition of the local-read protocol and the condition from the global-read protocol that there be no consistency constraints between local and global data.

The global-read–write/local-read protocol ensures strong correctness if all these conditions hold:

1. Local transactions may access local data items, and may read global data items stored at the site (although they must not write global data items).
2. Global transactions may access global data items as well as local data items (that is, they may read and write all data).
3. There are no consistency constraints between local and global data items.
4. No transaction may have a value dependency.

24.6.2 Ensuring Global Serializability

Early multidatabase systems restricted global transactions to be read only. They thus avoided the possibility of global transactions introducing inconsistency to the data, but were not sufficiently restrictive to ensure global serializability. It is indeed possible to get such global schedules and to develop a scheme to ensure global serializability, and we ask you to do both in Exercise 24.15.

There are a number of general schemes to ensure global serializability in an environment where update as well read-only transactions can execute. Several of these schemes are based on the idea of a **ticket**. A special data item called a ticket is created in each local database system. Every global transaction that accesses data at a site must write the ticket at that site. This requirement ensures that global transactions conflict directly at every site they visit. Furthermore, the global transaction manager can control the order in which global transactions are serialized, by controlling the order in which the tickets are accessed. References to such schemes appear in the bibliographical notes.

If we want to ensure global serializability in an environment where no direct local conflicts are generated in each site, some assumptions must be made about the schedules allowed by the local database system. For example, if the local schedules are such that the commit order and serialization order are always identical, we can ensure serializability by controlling only the order in which transactions commit.

The problem with schemes that ensure global serializability is that they may restrict concurrency unduly. They are particularly likely to do so because most transactions submit SQL statements to the underlying database system, instead of submitting individual **read**, **write**, **commit**, and **abort** steps. Although it is still possible to ensure global serializability under this assumption, the level of concurrency may be such that other schemes, such as the two-level serializability technique discussed in Section 24.6.1, are attractive alternatives.

24.7 Summary

- Workflows are activities that involve the coordinated execution of multiple tasks performed by different processing entities. They exist not just in computer applications, but also in almost all organizational activities. With the growth of networks, and the existence of multiple autonomous database systems, workflows provide a convenient way of carrying out tasks that involve multiple systems.

- Although the usual ACID transactional requirements are too strong or are unimplementable for such workflow applications, workflows must satisfy a limited set of transactional properties that guarantee that a process is not left in an inconsistent state.

- Transaction-processing monitors were initially developed as multithreaded servers that could service large numbers of terminals from a single process. They have since evolved, and today they provide the infrastructure for building and administering complex transaction-processing systems that have a large number of clients and multiple servers. They provide services such as durable queueing of client requests and server responses, routing of client messages to servers, persistent messaging, load balancing, and coordination of two-phase commit when transactions access multiple servers.

- Large main memories are exploited in certain systems to achieve high system throughput. In such systems, logging is a bottleneck. Under the group-commit concept, the number of outputs to stable storage can be reduced, thus releasing this bottleneck.

- The efficient management of long-duration interactive transactions is more complex, because of the long-duration waits, and because of the possibility of aborts. Since the concurrency-control techniques used in Chapter 16 use waits, aborts, or both, alternative techniques must be considered. These techniques must ensure correctness without requiring serializability.

- A long-duration transaction is represented as a nested transaction with atomic database operations at the lowest level. If a transaction fails, only active short-duration transactions abort. Active long-duration transactions resume once any short-duration transactions have recovered. A compensating transaction is needed to undo updates of nested transactions that have committed, if the outer-level transaction fails.

- In systems with real-time constraints, correctness of execution involves not only database consistency but also deadline satisfaction. The wide variance of execution times for read and write operations complicates the transaction-management problem for time-constrained systems.
- A multidatabase system provides an environment in which new database applications can access data from a variety of pre-existing databases located in various heterogeneous hardware and software environments.

  The local database systems may employ different logical models and data-definition and data-manipulation languages, and may differ in their concurrency-control and transaction-management mechanisms. A multidatabase system creates the illusion of logical database integration, without requiring physical database integration.

Review Terms

- TP monitor
- TP-monitor architectures
  - ☐ Process per client
  - ☐ Single server
  - ☐ Many server, single router
  - ☐ Many server, many router
- Multitasking
- Context switch
- Multithreaded server
- Queue manager
- Application coordination
  - ☐ Resource manager
  - ☐ Remote procedure call (RPC)
- Transactional Workflows
  - ☐ Task
  - ☐ Processing entity
  - ☐ Workflow specification
  - ☐ Workflow execution
- Workflow state
  - ☐ Execution states
  - ☐ Output values
  - ☐ External variables
- Workflow failure atomicity
- Workflow termination states
  - ☐ Acceptable
  - ☐ Nonacceptable
  - ☐ Committed
  - ☐ Aborted
- Workflow recovery
- Workflow-management system
- Workflow-management system architectures
  - ☐ Centralized
  - ☐ Partially distributed
  - ☐ Fully distributed
- Main-memory databases
- Group commit
- Real-time systems
- Deadlines
  - ☐ Hard deadline
  - ☐ Firm deadline
  - ☐ Soft deadline
- Real-time databases
- Long-duration transactions
- Exposure of uncommitted data
- Subtasks

- Nonserializable executions
- Nested transactions
- Multilevel transactions
- Saga
- Compensating transactions
- Logical logging
- Multidatabase systems
- Autonomy
- Local transactions
- Global transactions
- Two-level serializability (2LSR)
- Strong correctness
- Local data
- Global data
- Protocols
  - ☐ Global-read
  - ☐ Local-read
  - ☐ Value dependency
  - ☐ Global-read–write/local-read
- Ensuring global serializability
- Ticket

Exercises

24.1 Explain how a TP monitor manages memory and processor resources more effectively than a typical operating system.

24.2 Compare TP monitor features with those provided by Web servers supporting servlets (such servers have been nicknamed *TP-lite*).

24.3 Consider the process of admitting new students at your university (or new employees at your organization).

a. Give a high-level picture of the workflow starting from the student application procedure.
b. Indicate acceptable termination states, and which steps involve human intervention.
c. Indicate possible errors (including deadline expiry) and how they are dealt with.
d. Study how much of the workflow has been automated at your university.

24.4 Like database systems, workflow systems also require concurrency and recovery management. List three reasons why we cannot simply apply a relational database system using 2PL, physical undo logging, and 2PC.

24.5 If the entire database fits in main memory, do we still need a database system to manage the data? Explain your answer.

24.6 Consider a main-memory database system recovering from a system crash. Explain the relative merits of

- Loading the entire database back into main memory before resuming transaction processing
- Loading data as it is requested by transactions

24.7 In the group-commit technique, how many transactions should be part of a group? Explain your answer.

24.8 Is a high-performance transaction system necessarily a real-time system? Why or why not?

24.9 In a database system using write-ahead logging, what is the worst-case number of disk accesses required to read a data item? Explain why this presents a problem to designers of real-time database systems.

24.10 Explain why it may be impractical to require serializability for long-duration transactions.

24.11 Consider a multithreaded process that delivers messages from a durable queue of persistent messages. Different threads may run concurrently, attempting to deliver different messages. In case of a delivery failure, the message must be restored in the queue. Model the actions that each thread carries out as a multilevel transaction, so that locks on the queue need not be held till a message is delivered.

24.12 Discuss the modifications that need to be made in each of the recovery schemes covered in Chapter 17 if we allow nested transactions. Also, explain any differences that result if we allow multilevel transactions.

24.13 What is the purpose of compensating transactions? Present two examples of their use.

24.14 Consider a multidatabase system in which it is guaranteed that at most one global transaction is active at any time, and every local site ensures local serializability.

a. Suggest ways in which the multidatabase system can ensure that there is at most one active global transaction at any time.

b. Show by example that it is possible for a nonserializable global schedule to result despite the assumptions.

24.15 Consider a multidatabase system in which every local site ensures local serializability, and all global transactions are read only.

a. Show by example that nonserializable executions may result in such a system.

b. Show how you could use a ticket scheme to ensure global serializability.

Bibliographical Notes

Gray and Edwards [1995] provides an overview of TP monitor architectures; Gray and Reuter [1993] provides a detailed (and excellent) textbook description of transaction-processing systems, including chapters on TP monitors. Our description of TP monitors is modeled on these two sources. X/Open [1991] defines the X/Open XA interface. Transaction processing in Tuxedo is described in Huffman [1993]. Wipfler [1987] is one of several texts on application development using CICS.

Fischer [2001] is a handbook on workflow systems. A reference model for workflows, proposed by the Workflow Management Coalition, is presented in Hollinsworth

[1994]. The Web site of the coalition is www.wfmc.org. Our description of workflows follows the model of Rusinkiewicz and Sheth [1995].

Reuter [1989] presents ConTracts, a method for grouping transactions into multitransaction activities. Some issues related to workflows were addressed in the work on long-running activities described by Dayal et al. [1990] and Dayal et al. [1991]. The authors propose event–condition–action rules as a technique for specifying workflows. Jin et al. [1993] describes workflow issues in telecommunication applications.

Garcia-Molina and Salem [1992] provides an overview of main-memory databases. Jagadish et al. [1993] describes a recovery algorithm designed for main-memory databases. A storage manager for main-memory databases is described in Jagadish et al. [1994].

Transaction processing in real-time databases is discussed by Abbott and Garcia-Molina [1999] and Dayal et al. [1990]. Barclay et al. [1982] describes a real-time database system used in a telecommunications switching system. Complexity and correctness issues in real-time databases are addressed by Korth et al. [1990b] and Soparkar et al. [1995]. Concurrency control and scheduling in real-time databases are discussed by Haritsa et al. [1990], Hong et al. [1993], and Pang et al. [1995]. Ozsoyoglu and Snodgrass [1995] is a survey of research in real-time and temporal databases.

Nested and multilevel transactions are presented by Lynch [1983], Moss [1982], Moss [1985], Lynch and Merritt [1986], Fekete et al. [1990b], Fekete et al. [1990a], Korth and Speegle [1994], and Pu et al. [1988]. Theoretical aspects of multilevel transactions are presented in Lynch et al. [1988] and Weihl and Liskov [1990].

Several extended-transaction models have been defined including Sagas (Garcia-Molina and Salem [1987]), ACTA (Chrysanthis and Ramamritham [1994]), the ConTract model (Wachter and Reuter [1992]), ARIES (Mohan et al. [1992] and Rothermel and Mohan [1989]), and the NT/PV model (Korth and Speegle [1994]).

Splitting transactions to achieve higher performance is addressed in Shasha et al. [1995]. A model for concurrency in nested transactions systems is presented in Beeri et al. [1989]. Relaxation of serializability is discussed in Garcia-Molina [1983] and Sha et al. [1988]. Recovery in nested transaction systems is discussed by Moss [1987], Haerder and Rothermel [1987], Rothermel and Mohan [1989]. Multilevel transaction management is discussed in Weikum [1991].

Gray [1981], Skarra and Zdonik [1989], Korth and Speegle [1988], and Korth and Speegle [1990] discuss long-duration transactions. Transaction processing for long-duration transactions is considered by Weikum and Schek [1984], Haerder and Rothermel [1987], Weikum et al. [1990], and Korth et al. [1990a]. Salem et al. [1994] presents an extension of 2PL for long-duration transactions by allowing the early release of locks under certain circumstances. Transaction processing in design and software-engineering applications is discussed in Korth et al. [1988], Kaiser [1990], and Weikum [1991].

Transaction processing in multidatabase systems is discussed in Breitbart et al. [1990], Breitbart et al. [1991], Breitbart et al. [1992], Soparkar et al. [1991], Mehrotra et al. [1992b] and Mehrotra et al. [1992a]. The ticket scheme is presented in Georgakopoulos et al. [1994]. 2LSR is introduced in Mehrotra et al. [1991]. An earlier approach, called *quasi-serializability*, is presented in Du and Elmagarmid [1989].