Chapter 11
Distributed Concurrency Control

As we discussed in Chapter 10, concurrency control deals with the isolation and consistency properties of transactions. The distributed concurrency control mechanism of a distributed DBMS ensures that the consistency of the database, as defined in Section 10.2.2, is maintained in a multiuser distributed environment. If transactions are internally consistent (i.e., do not violate any consistency constraints), the simplest way of achieving this objective is to execute each transaction alone, one after another. It is obvious that such an alternative is only of theoretical interest and would not be implemented in any practical system, since it minimizes the system throughput. The level of concurrency (i.e., the number of concurrent transactions) is probably the most important parameter in distributed systems [Balter et al., 1982]. Therefore, the concurrency control mechanism attempts to find a suitable trade-off between maintaining the consistency of the database and maintaining a high level of concurrency.

In this chapter, we make two major assumptions: the distributed system is fully reliable and does not experience any failures (of hardware or software), and the database is not replicated. Even though these are unrealistic assumptions, they permit us to delineate the issues related to the management of concurrency from those related to the operation of a reliable distributed system and those related to maintaining replicas. In Chapter 12, we discuss how the algorithms that are presented in this chapter need to be enhanced to operate in an unreliable environment. In Chapter 13 we address the issues related to replica management.

We start our discussion of concurrency control with a presentation of serializability theory in Section 11.1. Serializability is the most widely accepted correctness criterion for concurrency control algorithms. In Section 11.2 we present a taxonomy of algorithms that will form the basis for most of the discussion in the remainder of the chapter. Sections 11.3 and 11.4 cover the two major classes of algorithms: locking-based and timestamp ordering-based. Both locking and timestamp ordering classes cover what is called pessimistic algorithms; optimistic concurrency control is discussed in Section 11.5. Any locking-based algorithm may result in deadlocks, requiring special management methods. Various deadlock management techniques are therefore the topic of Section 11.6. In Section 11.7, we discuss “relaxed” con-
currency control approaches. These are mechanisms which use weaker correctness criteria than serializability, or relax the isolation property of transactions.

### 11.1 Serializability Theory

In Section 10.1.3 we discussed the issue of isolating transactions from one another in terms of their effects on the database. We also pointed out that if the concurrent execution of transactions leaves the database in a state that can be achieved by their serial execution in some order, problems such as lost updates will be resolved. This is exactly the point of the serializability argument. The remainder of this section addresses serializability issues more formally.

A history $R$ (also called a schedule) is defined over a set of transactions $T = \{T_1, T_2, \ldots, T_n\}$ and specifies an interleaved order of execution of these transactions’ operations. Based on the definition of a transaction introduced in Section 10.1, the history can be specified as a partial order over $T$. We need a few preliminaries, though, before we present the formal definition.

Recall the definition of conflicting operations that we gave in Chapter 10. Two operations $O_{ij}(x)$ and $O_{kl}(x)$ ($i$ and $k$ representing transactions and are not necessarily distinct) accessing the same database entity $x$ are said to be in conflict if at least one of them is a write operation. Note two things in this definition. First, read operations do not conflict with each other. We can, therefore, talk about two types of conflicts: read-write (or write-read), and write-write. Second, the two operations can belong to the same transaction or to two different transactions. In the latter case, the two transactions are said to be conflicting. Intuitively, the existence of a conflict between two operations indicates that their order of execution is important. The ordering of two read operations is insignificant.

We first define a complete history, which defines the execution order of all operations in its domain. We will then define a history as a prefix of a complete history. Formally, a complete history $H^c_T$ defined over a set of transactions $T = \{T_1, T_2, \ldots, T_n\}$ is a partial order $H^c_T = (\Sigma_T, \prec_H)$ where

1. $\Sigma_T = \bigcup_{i=1}^n \Sigma_i$.
2. $\prec_H \supseteq \bigcup_{i=1}^n \prec_{T_i}$.
3. For any two conflicting operations $O_{ij}, O_{kl} \in \Sigma_T$, either $O_{ij} \prec_H O_{kl}$, or $O_{kl} \prec_H O_{ij}$.

The first condition simply states that the domain of the history is the union of the domains of individual transactions. The second condition defines the ordering relation of the history as a superset of the ordering relations of individual transactions. This maintains the ordering of operations within each transaction. The final condition simply defines the execution order among conflicting operations in $H$. 