The concept of transaction provides a mechanism for describing logical units of database processing. Transaction processing systems are systems with large databases and hundreds of concurrent users executing database transactions. Examples of such systems include airline reservations, banking, credit card processing, online retail purchasing, stock markets, supermarket checkouts, and many other applications. These systems require high availability and fast response time for hundreds of concurrent users. In this chapter we present the concepts that are needed in transaction processing systems. We define the concept of a transaction, which is used to represent a logical unit of database processing that must be completed in its entirety to ensure correctness. A transaction is typically implemented by a computer program, which includes database commands such as retrievals, insertions, deletions, and updates. We introduced some of the basic techniques for database programming in Chapters 13 and 14.

In this chapter, we focus on the basic concepts and theory that are needed to ensure the correct executions of transactions. We discuss the concurrency control problem, which occurs when multiple transactions submitted by various users interfere with one another in a way that produces incorrect results. We also discuss the problems that can occur when transactions fail, and how the database system can recover from various types of failures.

This chapter is organized as follows. Section 21.1 informally discusses why concurrency control and recovery are necessary in a database system. Section 21.2 defines the term transaction and discusses additional concepts related to transaction processing in database systems. Section 21.3 presents the important properties of atomicity, consistency preservation, isolation, and durability or permanency—called the
ACID properties—that are considered desirable in transaction processing systems. Section 21.4 introduces the concept of schedules (or histories) of executing transactions and characterizes the recoverability of schedules. Section 21.5 discusses the notion of serializability of concurrent transaction execution, which can be used to define correct execution sequences (or schedules) of concurrent transactions. In Section 21.6, we present some of the commands that support the transaction concept in SQL. Section 21.7 summarizes the chapter.

The two following chapters continue with more details on the actual methods and techniques used to support transaction processing. Chapter 22 gives an overview of the basic concurrency control protocols and Chapter 23 introduces recovery techniques.

21.1 Introduction to Transaction Processing

In this section we discuss the concepts of concurrent execution of transactions and recovery from transaction failures. Section 21.1.1 compares single-user and multiuser database systems and demonstrates how concurrent execution of transactions can take place in multiuser systems. Section 21.1.2 defines the concept of transaction and presents a simple model of transaction execution based on read and write database operations. This model is used as the basis for defining and formalizing concurrency control and recovery concepts. Section 21.1.3 uses informal examples to show why concurrency control techniques are needed in multiuser systems. Finally, Section 21.1.4 discusses why techniques are needed to handle recovery from system and transaction failures by discussing the different ways in which transactions can fail while executing.

21.1.1 Single-User versus Multiuser Systems

One criterion for classifying a database system is according to the number of users who can use the system concurrently. A DBMS is single-user if at most one user at a time can use the system, and it is multiuser if many users can use the system—and hence access the database—concurrently. Single-user DBMSs are mostly restricted to personal computer systems; most other DBMSs are multiuser. For example, an airline reservations system is used by hundreds of travel agents and reservation clerks concurrently. Database systems used in banks, insurance agencies, stock exchanges, supermarkets, and many other applications are multiuser systems. In these systems, hundreds or thousands of users are typically operating on the database by submitting transactions concurrently to the system.

Multiple users can access databases—and use computer systems—simultaneously because of the concept of multiprogramming, which allows the operating system of the computer to execute multiple programs—or processes—at the same time. A single central processing unit (CPU) can only execute at most one process at a time. However, multiprogramming operating systems execute some commands from one process, then suspend that process and execute some commands from the next
process, and so on. A process is resumed at the point where it was suspended whenever it gets its turn to use the CPU again. Hence, concurrent execution of processes is actually **interleaved**, as illustrated in Figure 21.1, which shows two processes, A and B, executing concurrently in an interleaved fashion. Interleaving keeps the CPU busy when a process requires an input or output (I/O) operation, such as reading a block from disk. The CPU is switched to execute another process rather than remaining idle during I/O time. Interleaving also prevents a long process from delaying other processes.

If the computer system has multiple hardware processors (CPUs), **parallel processing** of multiple processes is possible, as illustrated by processes C and D in Figure 21.1. Most of the theory concerning concurrency control in databases is developed in terms of **interleaved concurrency**, so for the remainder of this chapter we assume this model. In a multiuser DBMS, the stored data items are the primary resources that may be accessed concurrently by interactive users or application programs, which are constantly retrieving information from and modifying the database.

**21.1.2 Transactions, Database Items, Read and Write Operations, and DBMS Buffers**

A **transaction** is an executing program that forms a logical unit of database processing. A transaction includes one or more database access operations—these can include insertion, deletion, modification, or retrieval operations. The database operations that form a transaction can either be embedded within an application program or they can be specified interactively via a high-level query language such as SQL. One way of specifying the transaction boundaries is by specifying explicit **begin transaction** and **end transaction** statements in an application program; in this case, all database access operations between the two are considered as forming one transaction. A single application program may contain more than one transaction if it contains several transaction boundaries. If the database operations in a transaction do not update the database but only retrieve data, the transaction is called a **read-only transaction**; otherwise it is known as a **read-write transaction**.
The *database model* that is used to present transaction processing concepts is quite simple when compared to the data models that we discussed earlier in the book, such as the relational model or the object model. A database is basically represented as a collection of named data items. The size of a data item is called its *granularity*. A data item can be a database record, but it can also be a larger unit such as a whole *disk block*, or even a smaller unit such as an individual *field (attribute) value* of some record in the database. The transaction processing concepts we discuss are independent of the data item granularity (size) and apply to data items in general. Each data item has a unique name, but this name is not typically used by the programmer; rather, it is just a means to uniquely identify each data item. For example, if the data item granularity is one disk block, then the disk block address can be used as the data item name. Using this simplified database model, the basic database access operations that a transaction can include are as follows:

- **read_item**(X). Reads a database item named X into a program variable. To simplify our notation, we assume that the program variable is also named X.
- **write_item**(X). Writes the value of program variable X into the database item named X.

As we discussed in Chapter 17, the basic unit of data transfer from disk to main memory is one block. Executing a read_item**(X)** command includes the following steps:

1. Find the address of the disk block that contains item X.
2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
3. Copy item X from the buffer to the program variable named X.

Executing a write_item**(X)** command includes the following steps:

1. Find the address of the disk block that contains item X.
2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
3. Copy item X from the program variable named X into its correct location in the buffer.
4. Store the updated block from the buffer back to disk (either immediately or at some later point in time).

It is step 4 that actually updates the database on disk. In some cases the buffer is not immediately stored to disk, in case additional changes are to be made to the buffer. Usually, the decision about when to store a modified disk block whose contents are in a main memory buffer is handled by the recovery manager of the DBMS in cooperation with the underlying operating system. The DBMS will maintain in the database cache a number of data buffers in main memory. Each buffer typically holds the contents of one database disk block, which contains some of the database items being processed. When these buffers are all occupied, and additional database disk blocks must be copied into memory, some buffer replacement policy is used to
choose which of the current buffers is to be replaced. If the chosen buffer has been modified, it must be written back to disk before it is reused.

A transaction includes read_item and write_item operations to access and update the database. Figure 21.2 shows examples of two very simple transactions. The read-set of a transaction is the set of all items that the transaction reads, and the write-set is the set of all items that the transaction writes. For example, the read-set of $T_1$ in Figure 21.2 is $\{X, Y\}$ and its write-set is also $\{X, Y\}$.

Concurrency control and recovery mechanisms are mainly concerned with the database commands in a transaction. Transactions submitted by the various users may execute concurrently and may access and update the same database items. If this concurrent execution is uncontrolled, it may lead to problems, such as an inconsistent database. In the next section we informally introduce some of the problems that may occur.

### 21.1.3 Why Concurrency Control Is Needed

Several problems can occur when concurrent transactions execute in an uncontrolled manner. We illustrate some of these problems by referring to a much simplified airline reservations database in which a record is stored for each airline flight. Each record includes the number of reserved seats on that flight as a named (uniquely identifiable) data item, among other information. Figure 21.2(a) shows a transaction $T_1$ that transfers $N$ reservations from one flight whose number of reserved seats is stored in the database item named $X$ to another flight whose number of reserved seats is stored in the database item named $Y$. Figure 21.2(b) shows a simpler transaction $T_2$ that just reserves $M$ seats on the first flight ($X$) referenced in transaction $T_1$. To simplify our example, we do not show additional portions of the transactions, such as checking whether a flight has enough seats available before reserving additional seats.

---

<table>
<thead>
<tr>
<th>(a)</th>
<th>$T_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_item($X$);</td>
<td></td>
</tr>
<tr>
<td>$X := X - N$;</td>
<td></td>
</tr>
<tr>
<td>write_item($X$);</td>
<td></td>
</tr>
<tr>
<td>read_item($Y$);</td>
<td></td>
</tr>
<tr>
<td>$Y := Y + N$;</td>
<td></td>
</tr>
<tr>
<td>write_item($Y$);</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(b)</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_item($X$);</td>
<td></td>
</tr>
<tr>
<td>$X := X + M$;</td>
<td></td>
</tr>
<tr>
<td>write_item($X$);</td>
<td></td>
</tr>
</tbody>
</table>

---

1We will not discuss buffer replacement policies here because they are typically discussed in operating systems textbooks.

2A similar, more commonly used example assumes a bank database, with one transaction doing a transfer of funds from account $X$ to account $Y$ and the other transaction doing a deposit to account $X$. 
When a database access program is written, it has the flight number, flight date, and the number of seats to be booked as parameters; hence, the same program can be used to execute *many different transactions*, each with a different flight number, date, and number of seats to be booked. For concurrency control purposes, a transaction is a *particular execution* of a program on a specific date, flight, and number of seats. In Figure 21.2(a) and (b), the transactions $T_1$ and $T_2$ are *specific executions* of the programs that refer to the specific flights whose numbers of seats are stored in data items $X$ and $Y$ in the database. Next we discuss the types of problems we may encounter with these two simple transactions if they run concurrently.

**The Lost Update Problem.** This problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database items incorrect. Suppose that transactions $T_1$ and $T_2$ are submitted at approximately the same time, and suppose that their operations are interleaved as shown in Figure 21.3(a); then the final value of item $X$ is incorrect because $T_2$ reads the value of $X$ before $T_1$ changes it in the database, and hence the updated value resulting from $T_1$ is lost. For example, if $X = 80$ at the start (originally there were 80 reservations on the flight), $N = 5$ ($T_1$ transfers 5 seat reservations from the flight corresponding to $X$ to the flight corresponding to $Y$), and $M = 4$ ($T_2$ reserves 4 seats on $X$), the final result should be $X = 79$. However, in the interleaving of operations shown in Figure 21.3(a), it is $X = 84$ because the update in $T_1$ that removed the five seats from $X$ was lost.

**The Temporary Update (or Dirty Read) Problem.** This problem occurs when one transaction updates a database item and then the transaction fails for some reason (see Section 21.1.4). Meanwhile, the updated item is accessed (read) by another transaction before it is changed back to its original value. Figure 21.3(b) shows an example where $T_1$ updates item $X$ and then fails before completion, so the system must change $X$ back to its original value. Before it can do so, however, transaction $T_2$ reads the *temporary* value of $X$, which will not be recorded permanently in the database because of the failure of $T_1$. The value of item $X$ that is read by $T_2$ is called *dirty data* because it has been created by a transaction that has not completed and committed yet; hence, this problem is also known as the *dirty read problem*.

**The Incorrect Summary Problem.** If one transaction is calculating an aggregate summary function on a number of database items while other transactions are updating some of these items, the aggregate function may calculate some values before they are updated and others after they are updated. For example, suppose that a transaction $T_3$ is calculating the total number of reservations on all the flights; meanwhile, transaction $T_1$ is executing. If the interleaving of operations shown in Figure 21.3(c) occurs, the result of $T_3$ will be off by an amount $N$ because $T_3$ reads the value of $X$ *after* $N$ seats have been subtracted from it but reads the value of $Y$ *before* those $N$ seats have been added to it.
21.1 Introduction to Transaction Processing

### Figure 21.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

<table>
<thead>
<tr>
<th>Time</th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>(a)</strong></td>
<td>read_item($X$); $X := X - N$; write_item($X$); read_item($Y$); $Y := Y + N$; write_item($Y$);</td>
<td>read_item($X$); $X := X + M$; write_item($X$);</td>
</tr>
</tbody>
</table>

Transaction $T_1$ fails and must change the value of $X$ back to its old value; meanwhile $T_2$ has read the temporary incorrect value of $X$.

<table>
<thead>
<tr>
<th>Time</th>
<th>$T_1$</th>
<th>$T_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>(b)</strong></td>
<td>read_item($X$); $X := X - N$; write_item($X$); read_item($Y$);</td>
<td>$sum := 0$; read_item($A$); $sum := sum + A$;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\vdots$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\vdots$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$sum := sum + X$; read_item($Y$); $sum := sum + Y$;</td>
</tr>
</tbody>
</table>

$T_3$ reads $X$ after $N$ is subtracted and reads $Y$ before $N$ is added; a wrong summary is the result (off by $N$).

Item $X$ has an incorrect value because its update by $T_1$ is lost (overwritten).
The Unrepeatable Read Problem. Another problem that may occur is called unrepeatable read, where a transaction $T$ reads the same item twice and the item is changed by another transaction $T'$ between the two reads. Hence, $T$ receives different values for its two reads of the same item. This may occur, for example, if during an airline reservation transaction, a customer inquires about seat availability on several flights. When the customer decides on a particular flight, the transaction then reads the number of seats on that flight a second time before completing the reservation, and it may end up reading a different value for the item.

21.1.4 Why Recovery Is Needed

Whenever a transaction is submitted to a DBMS for execution, the system is responsible for making sure that either all the operations in the transaction are completed successfully and their effect is recorded permanently in the database, or that the transaction does not have any effect on the database or any other transactions. In the first case, the transaction is said to be committed, whereas in the second case, the transaction is aborted. The DBMS must not permit some operations of a transaction $T$ to be applied to the database while other operations of $T$ are not, because the whole transaction is a logical unit of database processing. If a transaction fails after executing some of its operations but before executing all of them, the operations already executed must be undone and have no lasting effect.

Types of Failures. Failures are generally classified as transaction, system, and media failures. There are several possible reasons for a transaction to fail in the middle of execution:

1. A computer failure (system crash). A hardware, software, or network error occurs in the computer system during transaction execution. Hardware crashes are usually media failures—for example, main memory failure.

2. A transaction or system error. Some operation in the transaction may cause it to fail, such as integer overflow or division by zero. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error. Additionally, the user may interrupt the transaction during its execution.

3. Local errors or exception conditions detected by the transaction. During transaction execution, certain conditions may occur that necessitate cancellation of the transaction. For example, data for the transaction may not be found. An exception condition, such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal, to be canceled. This exception could be programmed in the transaction itself, and in such a case would not be considered as a transaction failure.

---

3 In general, a transaction should be thoroughly tested to ensure that it does not have any bugs (logical programming errors).

4 Exception conditions, if programmed correctly, do not constitute transaction failures.
4. **Concurrency control enforcement.** The concurrency control method (see Chapter 22) may decide to abort a transaction because it violates serializability (see Section 21.5), or it may abort one or more transactions to resolve a state of deadlock among several transactions (see Section 22.1.3). Transactions aborted because of serializability violations or deadlocks are typically restarted automatically at a later time.

5. **Disk failure.** Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.

6. **Physical problems and catastrophes.** This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

Failures of types 1, 2, 3, and 4 are more common than those of types 5 or 6. Whenever a failure of type 1 through 4 occurs, the system must keep sufficient information to quickly recover from the failure. Disk failure or other catastrophic failures of type 5 or 6 do not happen frequently; if they do occur, recovery is a major task. We discuss recovery from failure in Chapter 23.

The concept of transaction is fundamental to many techniques for concurrency control and recovery from failures.

## 21.2 Transaction and System Concepts

In this section we discuss additional concepts relevant to transaction processing. Section 21.2.1 describes the various states a transaction can be in, and discusses other operations needed in transaction processing. Section 21.2.2 discusses the system log, which keeps information about transactions and data items that will be needed for recovery. Section 21.2.3 describes the concept of commit points of transactions, and why they are important in transaction processing.

### 21.2.1 Transaction States and Additional Operations

A transaction is an atomic unit of work that should either be completed in its entirety or not done at all. For recovery purposes, the system needs to keep track of when each transaction starts, terminates, and commits or aborts (see Section 21.2.3). Therefore, the recovery manager of the DBMS needs to keep track of the following operations:

- **BEGIN_TRANSACTION.** This marks the beginning of transaction execution.
- **READ or WRITE.** These specify read or write operations on the database items that are executed as part of a transaction.
- **END_TRANSACTION.** This specifies that READ and WRITE transaction operations have ended and marks the end of transaction execution. However, at this point it may be necessary to check whether the changes introduced by
the transaction can be permanently applied to the database (committed) or whether the transaction has to be aborted because it violates serializability (see Section 21.5) or for some other reason.

- COMMIT_TRANSACTION. This signals a successful end of the transaction so that any changes (updates) executed by the transaction can be safely committed to the database and will not be undone.

- ROLLBACK (or ABORT). This signals that the transaction has ended unsuccessfully, so that any changes or effects that the transaction may have applied to the database must be undone.

Figure 21.4 shows a state transition diagram that illustrates how a transaction moves through its execution states. A transaction goes into an active state immediately after it starts execution, where it can execute its READ and WRITE operations. When the transaction ends, it moves to the partially committed state. At this point, some recovery protocols need to ensure that a system failure will not result in an inability to record the changes of the transaction permanently (usually by recording changes in the system log, discussed in the next section). Once this check is successful, the transaction is said to have reached its commit point and enters the committed state. Commit points are discussed in more detail in Section 21.2.3. When a transaction is committed, it has concluded its execution successfully and all its changes must be recorded permanently in the database, even if a system failure occurs.

However, a transaction can go to the failed state if one of the checks fails or if the transaction is aborted during its active state. The transaction may then have to be rolled back to undo the effect of its WRITE operations on the database. The terminated state corresponds to the transaction leaving the system. The transaction information that is maintained in system tables while the transaction has been running is removed when the transaction terminates. Failed or aborted transactions may be restarted later—either automatically or after being resubmitted by the user—as brand new transactions.

---

5Optimistic concurrency control (see Section 22.4) also requires that certain checks are made at this point to ensure that the transaction did not interfere with other executing transactions.
21.2.2 The System Log

To be able to recover from failures that affect transactions, the system maintains a log\textsuperscript{6} to keep track of all transaction operations that affect the values of database items, as well as other transaction information that may be needed to permit recovery from failures. The log is a sequential, append-only file that is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure. Typically, one (or more) main memory buffers hold the last part of the log file, so that log entries are first added to the main memory buffer. When the log buffer is filled, or when certain other conditions occur, the log buffer is appended to the end of the log file on disk. In addition, the log file from disk is periodically backed up to archival storage (tape) to guard against catastrophic failures. The following are the types of entries—called log records—that are written to the log file and the corresponding action for each log record. In these entries, $T$ refers to a unique transaction-id that is generated automatically by the system for each transaction and that is used to identify each transaction:

1. \texttt{[start\_transaction, T]}. Indicates that transaction $T$ has started execution.
2. \texttt{[write\_item, T, X, old\_value, new\_value]}. Indicates that transaction $T$ has changed the value of database item $X$ from $old\_value$ to $new\_value$.
3. \texttt{[read\_item, T, X]}. Indicates that transaction $T$ has read the value of database item $X$.
4. \texttt{[commit, T]}. Indicates that transaction $T$ has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
5. \texttt{[abort, T]}. Indicates that transaction $T$ has been aborted.

Protocols for recovery that avoid cascading rollbacks (see Section 21.4.2)—which include nearly all practical protocols—do not require that READ operations are written to the system log. However, if the log is also used for other purposes—such as auditing (keeping track of all database operations)—then such entries can be included. Additionally, some recovery protocols require simpler WRITE entries only include one of $new\_value$ and $old\_value$ instead of including both (see Section 21.4.2).

Notice that we are assuming that all permanent changes to the database occur within transactions, so the notion of recovery from a transaction failure amounts to either undoing or redoing transaction operations individually from the log. If the system crashes, we can recover to a consistent database state by examining the log and using one of the techniques described in Chapter 23. Because the log contains a record of every WRITE operation that changes the value of some database item, it is possible to undo the effect of these WRITE operations of a transaction $T$ by tracing backward through the log and resetting all items changed by a WRITE operation of $T$ to their old_values. Redo of an operation may also be necessary if a transaction has its updates recorded in the log but a failure occurs before the system can be sure that

\textsuperscript{6}The log has sometimes been called the DBMS journal.
Chapter 21  Introduction to Transaction Processing Concepts and Theory

all these new_values have been written to the actual database on disk from the main memory buffers.\(^7\)

### 21.2.3 Commit Point of a Transaction

A transaction \( T \) reaches its **commit point** when all its operations that access the database have been executed successfully and the effect of all the transaction operations on the database have been recorded in the log. Beyond the commit point, the transaction is said to be **committed**, and its effect must be permanently recorded in the database. The transaction then writes a commit record \([\text{commit}, T]\) into the log. If a system failure occurs, we can search back in the log for all transactions \( T \) that have written a \([\text{start\_transaction}, T]\) record into the log but have not written their \([\text{commit}, T]\) record yet; these transactions may have to be rolled back to undo their effect on the database during the recovery process. Transactions that have written their commit record in the log must also have recorded all their WRITE operations in the log, so their effect on the database can be redone from the log records.

Notice that the log file must be kept on disk. As discussed in Chapter 17, updating a disk file involves copying the appropriate block of the file from disk to a buffer in main memory, updating the buffer in main memory, and copying the buffer to disk. It is common to keep one or more blocks of the log file in main memory buffers, called the **log buffer**, until they are filled with log entries and then to write them back to disk only once, rather than writing to disk every time a log entry is added. This saves the overhead of multiple disk writes of the same log file buffer. At the time of a system crash, only the log entries that have been written back to disk are considered in the recovery process because the contents of main memory may be lost. Hence, **before** a transaction reaches its commit point, any portion of the log that has not been written to the disk yet must now be written to the disk. This process is called **force-writing** the log buffer before committing a transaction.

### 21.3 Desirable Properties of Transactions

Transactions should possess several properties, often called the **ACID** properties; they should be enforced by the concurrency control and recovery methods of the DBMS. The following are the ACID properties:

- **Atomicity.** A transaction is an atomic unit of processing; it should either be performed in its entirety or not performed at all.

- **Consistency preservation.** A transaction should be consistency preserving, meaning that if it is completely executed from beginning to end without interference from other transactions, it should take the database from one consistent state to another.

- **Isolation.** A transaction should appear as though it is being executed in isolation from other transactions, even though many transactions are executing

\(^7\)Undo and redo are discussed more fully in Chapter 23.
concurrently. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.

- **Durability or permanency.** The changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure.

The atomicity property requires that we execute a transaction to completion. It is the responsibility of the transaction recovery subsystem of a DBMS to ensure atomicity. If a transaction fails to complete for some reason, such as a system crash in the midst of transaction execution, the recovery technique must undo any effects of the transaction on the database. On the other hand, write operations of a committed transaction must be eventually written to disk.

The preservation of consistency is generally considered to be the responsibility of the programmers who write the database programs or of the DBMS module that enforces integrity constraints. Recall that a database state is a collection of all the stored data items (values) in the database at a given point in time. A consistent state of the database satisfies the constraints specified in the schema as well as any other constraints on the database that should hold. A database program should be written in a way that guarantees that, if the database is in a consistent state before executing the transaction, it will be in a consistent state after the complete execution of the transaction, assuming that no interference with other transactions occurs.

The isolation property is enforced by the concurrency control subsystem of the DBMS. If every transaction does not make its updates (write operations) visible to other transactions until it is committed, one form of isolation is enforced that solves the temporary update problem and eliminates cascading rollbacks (see Chapter 23) but does not eliminate all other problems. There have been attempts to define the level of isolation of a transaction. A transaction is said to have level 0 (zero) isolation if it does not overwrite the dirty reads of higher-level transactions. Level 1 (one) isolation has no lost updates, and level 2 isolation has no lost updates and no dirty reads. Finally, level 3 isolation (also called true isolation) has, in addition to level 2 properties, repeatable reads.

And last, the durability property is the responsibility of the recovery subsystem of the DBMS. We will introduce how recovery protocols enforce durability and atomicity in the next section and then discuss this in more detail in Chapter 23.

### 21.4 Characterizing Schedules Based on Recoverability

When transactions are executing concurrently in an interleaved fashion, then the order of execution of operations from all the various transactions is known as a schedule (or history). In this section, first we define the concept of schedules, and

---

8We will discuss concurrency control protocols in Chapter 22.

9The SQL syntax for isolation level discussed later in Section 21.6 is closely related to these levels.