Chapter 10
Introduction to Transaction Management

Up to this point the basic access primitive that we have considered has been a query. Our focus has been on retrieve-only (or read-only) queries that read data from a distributed database. We have not yet considered what happens if, for example, two queries attempt to update the same data item, or if a system failure occurs during execution of a query. For retrieve-only queries, neither of these conditions is a problem. One can have two queries reading the value of the same data item concurrently. Similarly, a read-only query can simply be restarted after a system failure is handled. On the other hand, it is not difficult to see that for update queries, these conditions can have disastrous effects on the database. We cannot, for example, simply restart the execution of an update query following a system failure since certain data item values may already have been updated prior to the failure and should not be updated again when the query is restarted. Otherwise, the database would contain incorrect data.

The fundamental point here is that there is no notion of "consistent execution" or "reliable computation" associated with the concept of a query. The concept of a *transaction* is used in database systems as a basic unit of consistent and reliable computing. Thus queries are executed as transactions once their execution strategies are determined and they are translated into primitive database operations.

In the discussion above, we used the terms *consistent* and *reliable* quite informally. Due to their importance in our discussion, we need to define them more precisely. We differentiate between *database consistency* and *transaction consistency*.

A database is in a *consistent state* if it obeys all of the consistency (integrity) constraints defined over it (see Chapter 5). State changes occur due to modifications, insertions, and deletions (together called *updates*). Of course, we want to ensure that the database never enters an inconsistent state. Note that the database can be (and usually is) temporarily inconsistent during the execution of a transaction. The important point is that the database should be consistent when the transaction terminates (Figure 10.1).

Transaction consistency, on the other hand, refers to the actions of concurrent transactions. We would like the database to remain in a consistent state even if there are a number of user requests that are concurrently accessing (reading or updating)

M.T. Özsu and P. Valduriez, *Principles of Distributed Database Systems: Third Edition*,
DOI 10.1007/978-1-4419-8834-8_10, © Springer Science+Business Media, LLC 2011

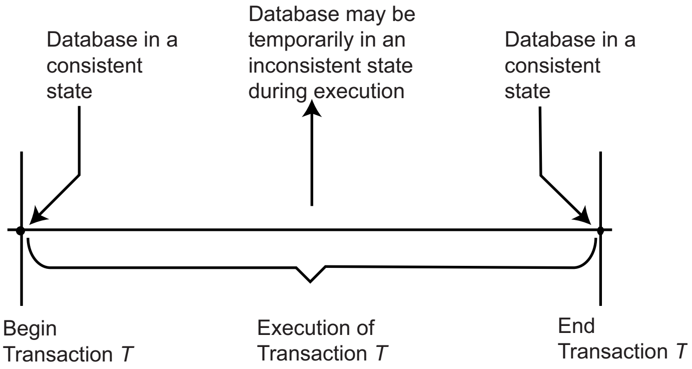
Database in a consistent state
Database may be temporarily in an inconsistent state during execution
Database in a consistent state
Begin Transaction *T*
Execution of Transaction *T*
End Transaction *T*

Fig. 10.1 A Transaction Model

the database. A complication arises when replicated databases are considered. A replicated database is in a *mutually consistent state* if all the copies of every data item in it have identical values. This is referred to as *one-copy equivalence* since all replica copies are forced to assume the same state at the end of a transaction's execution. There are more relaxed notions of replica consistency that allow replica values to diverge. These will be discussed later in Chapter 13.

Reliability refers to both the *resiliency* of a system to various types of failures and its capability to *recover* from them. A resilient system is tolerant of system failures and can continue to provide services even when failures occur. A recoverable DBMS is one that can get to a consistent state (by moving back to a previous consistent state or forward to a new consistent state) following various types of failures.

Transaction management deals with the problems of always keeping the database in a consistent state even when concurrent accesses and failures occur. In the upcoming two chapters, we investigate the issues related to managing transactions. A third chapter will address issues related to keeping replicated databases consistent. The purpose of the current chapter is to define the fundamental terms and to provide the framework within which these issues can be discussed. It also serves as a concise introduction to the problem and the related issues. We will therefore discuss the concepts at a high level of abstraction and will not present any management techniques.

The organization of this chapter is as follows. In the next section we formally and intuitively define the concept of a transaction. In Section 10.2 we discuss the properties of transactions and what the implications of each of these properties are in terms of transaction management. In Section 10.3 we present various types of transactions. In Section 10.4 we revisit the architectural model defined in Chapter 1 and indicate the modifications that are necessary to support transaction management.

10.1 Definition of a Transaction

Gray [1981] indicates that the transaction concept has its roots in contract law. He states, “In making a contract, two or more parties negotiate for a while and then make a deal. The deal is made binding by the joint signature of a document or by some other act (as simple as a handshake or a nod). If the parties are rather suspicious of one another or just want to be safe, they appoint an intermediary (usually called an escrow officer) to coordinate the commitment of the transaction.” The nice aspect of this historical perspective is that it does indeed encompass *some* of the fundamental properties of a transaction (atomicity and durability) as the term is used in database systems. It also serves to indicate the differences between a transaction and a query.

As indicated before, a transaction is a unit of consistent and reliable computation. Thus, intuitively, a transaction takes a database, performs an action on it, and generates a new version of the database, causing a state transition. This is similar to what a query does, except that if the database was consistent before the execution of the transaction, we can now guarantee that it will be consistent at the end of its execution regardless of the fact that (1) the transaction may have been executed concurrently with others, and (2) failures may have occurred during its execution.

In general, a transaction is considered to be made up of a sequence of read and write operations on the database, together with computation steps. In that sense, a transaction may be thought of as a program with embedded database access queries [Papadimitriou, 1986]. Another definition of a transaction is that it is a single execution of a program [Ullman, 1988]. A single query can also be thought of as a program that can be posed as a transaction.

Example 10.1. Consider the following SQL query for increasing by 10% the budget of the CAD/CAM project that we discussed (in Example 5.20):

```
UPDATE PROJ
SET    BUDGET = BUDGET*1.1
WHERE  PNAME= "CAD/CAM"
```

This query can be specified, using the embedded SQL notation, as a transaction by giving it a name (e.g., BUDGET_UPDATE) and declaring it as follows:

```
Begin_transaction BUDGET_UPDATE
begin
  EXEC SQL      UPDATE   PROJ
        SET     BUDGET = BUDGET*1.1
        WHERE   PNAME= “CAD/CAM”
end.
```

♦

The **Begin_transaction** and **end** statements delimit a transaction. Note that the use of delimiters is not enforced in every DBMS. If delimiters are not specified, a DBMS may simply treat as a transaction the entire program that performs a database access.

Example 10.2. In our discussion of transaction management concepts, we will use an airline reservation system example instead of the one used in the first nine chapters. The real-life implementation of this application almost always makes use of the transaction concept. Let us assume that there is a FLIGHT relation that records the data about each flight, a CUST relation for the customers who book flights, and an FC relation indicating which customers are on what flights. Let us also assume that the relation definitions are as follows (where the underlined attributes constitute the keys):

FLIGHT(FNO, DATE, SRC, DEST, STSOLD, CAP)
CUST(CNAME, ADDR, BAL)
FC(FNO, DATE, CNAME, SPECIAL)

The definition of the attributes in this database schema are as follows: FNO is the flight number, DATE denotes the flight date, SRC and DEST indicate the source and destination for the flight, STSOLD indicates the number of seats that have been sold on that flight, CAP denotes the passenger capacity on the flight, CNAME indicates the customer name whose address is stored in ADDR and whose account balance is in BAL, and SPECIAL corresponds to any special requests that the customer may have for a booking.

Let us consider a simplified version of a typical reservation application, where a travel agent enters the flight number, the date, and a customer name, and asks for a reservation. The transaction to perform this function can be implemented as follows, where database accesses are specified in embedded SQL notation:

```
Begin_transaction Reservation
begin
    input(flight_no, date, customer_name);                       (1)
    EXEC SQL UPDATE FLIGHT                                       (2)
             SET     STSOLD = STSOLD + 1
             WHERE   FNO = flight_no
             AND     DATE = date;
    EXEC SQL INSERT                                              (3)
             INTO    FC(FNO,DATE,CNAME,SPECIAL)
             VALUES (flight_no,date,customer_name, null);
    output("reservation completed")                              (4)
end.
```

Let us explain this example. First a point about notation. Even though we use embedded SQL, we do not follow its syntax very strictly. The lowercase terms are the program variables; the uppercase terms denote database relations and attributes as well as the SQL statements. Numeric constants are used as they are, whereas character constants are enclosed in quotes. Keywords of the host language are written in boldface, and *null* is a keyword for the null string.

The first thing that the transaction does [line (1)], is to input the flight number, the date, and the customer name. Line (2) updates the number of sold seats on the requested flight by one. Line (3) inserts a tuple into the FC relation. Here we assume that the customer is an old one, so it is not necessary to have an insertion into the CUST relation, creating a record for the client. The keyword *null* in line (3) indicates that the customer has no special requests on this flight. Finally, line (4) reports the result of the transaction to the agent's terminal. ♦

10.1.1 Termination Conditions of Transactions

The reservation transaction of Example 10.2 has an implicit assumption about its termination. It assumes that there will always be a free seat and does not take into consideration the fact that the transaction may fail due to lack of seats. This is an unrealistic assumption that brings up the issue of termination possibilities of transactions.

A transaction always terminates, even when there are failures as we will see in Chapter 12. If the transaction can complete its task successfully, we say that the transaction *commits*. If, on the other hand, a transaction stops without completing its task, we say that it *aborts*. Transactions may abort for a number of reasons, which are discussed in the upcoming chapters. In our example, a transaction aborts itself because of a condition that would prevent it from completing its task successfully. Additionally, the DBMS may abort a transaction due to, for example, deadlocks or other conditions. When a transaction is aborted, its execution is stopped and all of its already executed actions are *undone* by returning the database to the state before their execution. This is also known as *rollback*.

The importance of commit is twofold. The commit command signals to the DBMS that the effects of that transaction should now be reflected in the database, thereby making it visible to other transactions that may access the same data items. Second, the point at which a transaction is committed is a "point of no return." The results of the committed transaction are now *permanently* stored in the database and cannot be undone. The implementation of the commit command is discussed in Chapter 12.

Example 10.3. Let us return to our reservation system example. One thing we did not consider is that there may not be any free seats available on the desired flight. To cover this possibility, the reservation transaction needs to be revised as follows:

```
Begin_transaction Reservation
begin
    input(flight_no, date, customer_name);
    EXEC SQL SELECT STSOLD,CAP
             INTO   temp1,temp2
             FROM   FLIGHT
             WHERE  FNO = flight_no
             AND    DATE = date;
```

```
        if temp1 = temp2 then
        begin
            output("no free seats");
            Abort
        end
        else begin
            EXEC SQL UPDATE FLIGHT
                              SET      STSOLD = STSOLD + 1
                              WHERE  FNO = flight_no
                              AND      DATE = date;
            EXEC SQL INSERT
                              INTO     FC(FNO,DATE,CNAME,SPECIAL)
                              VALUES (flight_no, date, customer_name, null);
            Commit;
            output("reservation completed")
            end
        end-if
    end.
```

In this version the first SQL statement gets the STSOLD and CAP into the two variables temp1 and temp2. These two values are then compared to determine if any seats are available. The transaction either aborts if there are no free seats, or updates the STSOLD value and inserts a new tuple into the FC relation to represent the seat that was sold. ♦

Several things are important in this example. One is, obviously, the fact that if no free seats are available, the transaction is aborted[1]. The second is the ordering of the output to the user with respect to the abort and commit commands. Transactions can be aborted either due to application logic, as is the case here, or due to deadlocks or system failures. If the transaction is aborted, the user can be notified before the DBMS is instructed to abort it. However, in case of commit, the user notification has to follow the successful servicing (by the DBMS) of the commit command, for reliability reasons. These are discussed further in Section 10.2.4 and in Chapter 12.

10.1.2 Characterization of Transactions

Observe in the preceding examples that transactions read and write some data. This has been used as the basis for characterizing a transaction. The data items that a transaction reads are said to constitute its *read set* (*RS*). Similarly, the data items that a transaction writes are said to constitute its *write set* (*WS*). The read set and write

[1] We will be kind to the airlines and assume that they never overbook. Thus our reservation transaction does not need to check for that condition.

set of a transaction need not be mutually exclusive. The union of the read set and write set of a transaction constitutes its *base set* ($BS = RS \cup WS$).

Example 10.4. Considering the reservation transaction as specified in Example 10.3 and the insert to be a number of write operations, the above-mentioned sets are defined as follows:

$$\begin{aligned} RS[\text{Reservation}] &= \{\text{FLIGHT.STSOLD, FLIGHT.CAP}\} \\ WS[\text{Reservation}] &= \{\text{FLIGHT.STSOLD, FC.FNO, FC.DATE,} \\ &\qquad \text{FC.CNAME, FC.SPECIAL}\} \\ BS[\text{Reservation}] &= \{\text{FLIGHT.STSOLD, FLIGHT.CAP,} \\ &\qquad \text{FC.FNO, FC.DATE, FC.CNAME, FC.SPECIAL}\} \end{aligned}$$

Note that it may be appropriate to include FLIGHT.FNO and FLIGHT.DATE in the read set of Reservation since they are accessed during execution of the SQL query. We omit them to simplify the example. ♦

We have characterized transactions only on the basis of their read and write operations, without considering the insertion and deletion operations. We therefore base our discussion of transaction management concepts on *static* databases that do not grow or shrink. This simplification is made in the interest of simplicity. Dynamic databases have to deal with the problem of *phantoms*, which can be explained using the following example. Consider that transaction T_1, during its execution, searches the FC table for the names of customers who have ordered a special meal. It gets a set of CNAME for customers who satisfy the search criteria. While T_1 is executing, transaction T_2 inserts new tuples into FC with the special meal request, and commits. If T_1 were to re-issue the same search query later in its execution, it will get back a set of CNAME that is different than the original set it had retrieved. Thus, "phantom" tuples have appeared in the database. We do not discuss phantoms any further in this book; the topic is discussed at length by Eswaran et al. [1976] and Bernstein et al. [1987].

We should also point out that the read and write operations to which we refer are abstract operations that do not have one-to-one correspondence to physical I/O primitives. One read in our characterization may translate into a number of primitive read operations to access the index structures and the physical data pages. The reader should treat each read and write as a language primitive rather than as an operating system primitive.

10.1.3 Formalization of the Transaction Concept

By now, the meaning of a transaction should be intuitively clear. To reason about transactions and about the correctness of the management algorithms, it is necessary to define the concept formally. We denote by $O_{ij}(x)$ some *operation* O_j of transaction T_i that operates on a database entity x. Following the conventions adopted in the preceding section, $O_{ij} \in \{\text{read, write}\}$. Operations are assumed to be *atomic* (i.e., each is executed as an indivisible unit). We let OS_i denote the set of all operations in T_i (i.e., $OS_i = \bigcup_j O_{ij}$). We denote by N_i the termination condition for T_i, where $N_i \in \{\text{abort, commit}\}$[2].

With this terminology we can define a transaction T_i as a partial ordering over its operations and the termination condition. A partial order $P = \{\Sigma, \prec\}$ defines an ordering among the elements of Σ (called the *domain*) according to an irreflexive and transitive binary relation $\prec$ defined over Σ. In our case Σ consists of the operations and termination condition of a transaction, whereas $\prec$ indicates the execution order of these operations (which we will read as "precedes in execution order"). Formally, then, a transaction T_i is a partial order $T_i = \{\Sigma_i, \prec_i\}$, where

1. $\Sigma_i = OS_i \cup \{N_i\}$.
2. For any two operations $O_{ij}, O_{ik} \in OS_i$, if $O_{ij} = \{R(x) \text{or}\, W(x)\}$ and $O_{ik} = W(x)$ for any data item x, then either $O_{ij} \prec_i O_{ik}$ or $O_{ik} \prec_i O_{ij}$.
3. $\forall O_{ij} \in OS_i, O_{ij} \prec_i N_i$.

The first condition formally defines the domain as the set of read and write operations that make up the transaction, plus the termination condition, which may be either commit or abort. The second condition specifies the ordering relation between the conflicting read and write operations of the transaction, while the final condition indicates that the termination condition always follows all other operations.

There are two important points about this definition. First, the ordering relation $\prec$ is given and the definition does not attempt to construct it. The ordering relation is actually application dependent. Second, condition two indicates that the ordering between conflicting operations has to exist within $\prec$. Two operations, $O_i(x)$ and $O_j(x)$, are said to be in *conflict* if O_i = Write or O_j = Write (i.e., at least one of them is a Write and they access the same data item).

Example 10.5. Consider a simple transaction T that consists of the following steps:

Read(x)
Read(y)
$x \leftarrow x + y$
Write(x)
Commit

The specification of this transaction according to the formal notation that we have introduced is as follows:

$$\Sigma = \{R(x), R(y), W(x), C\}$$
$$\prec\, = \{(R(x), W(x)), (R(y), W(x)), (W(x), C), (R(x), C), (R(y), C)\}$$

where (O_i, O_j) as an element of the $\prec$ relation indicates that $O_i \prec O_j$. ♦

[2] From now on, we use the abbreviations R, W, A and C for the Read, Write, Abort, and Commit operations, respectively.

Notice that the ordering relation specifies the relative ordering of all operations with respect to the termination condition. This is due to the third condition of transaction definition. Also note that we do not specify the ordering between every pair of operations. That is why it is a *partial* order.

Example 10.6. The reservation transaction developed in Example 10.3 is more complex. Notice that there are two possible termination conditions, depending on the availability of seats. It might first seem that this is a contradiction of the definition of a transaction, which indicates that there can be only one termination condition. However, remember that a transaction is the execution of a program. It is clear that in any execution, only one of the two termination conditions can occur. Therefore, what exists is one transaction that aborts and another one that commits. Using this formal notation, the former can be specified as follows:

$$\begin{aligned}\Sigma &= \{R(\text{STSOLD}), R(\text{CAP}), A\}\\ \prec &= \{(O_1,A),(O_2,A)\}\end{aligned}$$

and the latter can be specified as

$$\begin{aligned}\Sigma = \{&R(\text{STSOLD}), R(\text{CAP}), W(\text{STSOLD}),\\ &W(\text{FNO}), W(\text{DATE}), W(\text{CNAME}), W(\text{SPECIAL}), C\}\\ \prec = \{&(O_1,O_3),(O_2,O_3),(O_1,O_4),(O_1,O_5),(O_1,O_6),(O_1,O_7),(O_2,O_4),\\ &(O_2,O_5),(O_2,O_6),(O_2,O_7),(O_1,C),(O_2,C),(O_3,C),(O_4,C),\\ &(O_5,C),(O_6,C),(O_7,C)\}\end{aligned}$$

where $O_1 = R(\text{STSOLD})$, $O_2 = R(\text{CAP})$, $O_3 = W(\text{STSOLD})$, $O_4 = W(\text{FNO})$, $O_5 = W(\text{DATE})$, $O_6 = W(\text{CNAME})$, and $O_7 = W(\text{SPECIAL})$. ♦

One advantage of defining a transaction as a partial order is its correspondence to a directed acyclic graph (DAG). Thus a transaction can be specified as a DAG whose vertices are the operations of a transaction and whose arcs indicate the ordering relationship between a given pair of operations. This will be useful in discussing the concurrent execution of a number of transactions (Chapter 11) and in arguing about their correctness by means of graph-theoretic tools.

Example 10.7. The transaction discussed in Example 10.5 can be represented as a DAG as depicted in Figure 10.2. Note that we do not draw the arcs that are implied by transitivity even though we indicate them as elements of $\prec$. ♦

In most cases we do not need to refer to the domain of the partial order separately from the ordering relation. Therefore, it is common to drop Σ from the transaction definition and use the name of the partial order to refer to both the domain and the name of the partial order. This is convenient since it allows us to specify the ordering of the operations of a transaction in a more straightforward manner by making use of their relative ordering in the transaction definition. For example, we can define the transaction of Example 10.5 as follows:

$$T = \{R(x), R(y), W(x), C\}$$

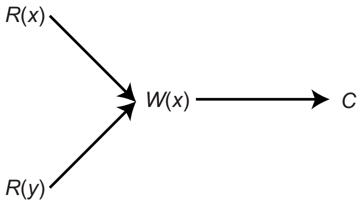

Fig. 10.2 DAG Representation of a Transaction

instead of the longer specification given before. We will therefore use the modified definition in this and subsequent chapters.

10.2 Properties of Transactions

The previous discussion clarifies the concept of a transaction. However, we have not yet provided any justification of our earlier claim that it is a unit of consistent and reliable computation. We do that in this section. The consistency and reliability aspects of transactions are due to four properties: (1) atomicity, (2) consistency, (3) isolation, and (4) durability. Together, these are commonly referred to as the ACID properties of transactions. They are not entirely independent of each other; usually there are dependencies among them as we will indicate below. We discuss each of these properties in the following sections.

10.2.1 Atomicity

Atomicity refers to the fact that a transaction is treated as a unit of operation. Therefore, either all the transaction's actions are completed, or none of them are. This is also known as the "all-or-nothing property." Notice that we have just extended the concept of atomicity from individual operations to the entire transaction. Atomicity requires that if the execution of a transaction is interrupted by any sort of failure, the DBMS will be responsible for determining what to do with the transaction upon recovery from the failure. There are, of course, two possible courses of action: it can either be terminated by completing the remaining actions, or it can be terminated by undoing all the actions that have already been executed.

One can generally talk about two types of failures. A transaction itself may fail due to input data errors, deadlocks, or other factors. In these cases either the transaction aborts itself, as we have seen in Example 10.2, or the DBMS may abort it while handling deadlocks, for example. Maintaining transaction atomicity in the presence of this type of failure is commonly called the *transaction recovery*. The second type of failure is caused by system crashes, such as media failures, processor failures, communication link breakages, power outages, and so on. Ensuring transaction atomicity in the presence of system crashes is called *crash recovery*. An important difference between the two types of failures is that during some types of system crashes, the information in volatile storage may be lost or inaccessible. Both types of recovery are parts of the reliability issue, which we discuss in considerable detail in Chapter 12.

10.2.2 Consistency

The *consistency* of a transaction is simply its correctness. In other words, a transaction is a correct program that maps one consistent database state to another. Verifying that transactions are consistent is the concern of integrity enforcement, covered in Chapter 5. Ensuring transaction consistency as defined at the beginning of this chapter, on the other hand, is the objective of concurrency control mechanisms, which we discuss in Chapter 11.

There is an interesting classification of consistency that parallels our discussion above and is equally important. This classification groups databases into four levels of consistency [Gray et al., 1976]. In the following definition (which is taken verbatim from the original paper), *dirty* data refers to data values that have been updated by a transaction prior to its commitment. Then, based on the concept of dirty data, the four levels are defined as follows:

> "Degree 3: Transaction T sees *degree 3 consistency* if:
>
> **1.** T does not overwrite dirty data of other transactions.
>
> **2.** T does not commit any writes until it completes all its writes [i.e., until the end of transaction (EOT)].
>
> **3.** T does not read dirty data from other transactions.
>
> **4.** Other transactions do not dirty any data read by T before T completes.
>
> Degree 2: Transaction T sees *degree 2 consistency* if:
>
> **1.** T does not overwrite dirty data of other transactions.
>
> **2.** T does not commit any writes before EOT.
>
> **3.** T does not read dirty data from other transactions.
>
> Degree 1: Transaction T sees *degree 1 consistency* if:
>
> **1.** T does not overwrite dirty data of other transactions.
>
> **2.** T does not commit any writes before EOT.

Degree 0: Transaction T sees *degree 0 consistency* if:

1. T does not overwrite dirty data of other transactions."

Of course, it is true that a higher degree of consistency encompasses all the lower degrees. The point in defining multiple levels of consistency is to provide application programmers the flexibility to define transactions that operate at different levels. Consequently, while some transactions operate at Degree 3 consistency level, others may operate at lower levels and may see, for example, dirty data.

10.2.3 Isolation

Isolation is the property of transactions that requires each transaction to see a consistent database at all times. In other words, an executing transaction cannot reveal its results to other concurrent transactions before its commitment.

There are a number of reasons for insisting on isolation. One has to do with maintaining the interconsistency of transactions. If two concurrent transactions access a data item that is being updated by one of them, it is not possible to guarantee that the second will read the correct value.

Example 10.8. Consider the following two concurrent transactions (T_1 and T_2), both of which access data item x. Assume that the value of x before they start executing is 50.

```
T1: Read(x)        T2: Read(x)
    x ← x + 1          x ← x + 1
    Write(x)           Write(x)
    Commit             Commit
```

The following is one possible sequence of execution of the actions of these transactions:

T_1: Read(x)
T_1: $x \leftarrow x+1$
T_1: Write(x)
T_1: Commit
T_2: Read(x)
T_2: $x \leftarrow x+1$
T_2: Write(x)
T_2: Commit

In this case, there are no problems; transactions T_1 and T_2 are executed one after the other and transaction T_2 reads 51 as the value of x. Note that if, instead, T_2 executes before T_1, T_2 reads 51 as the value of x. So, if T_1 and T_2 are executed one after the other (regardless of the order), the second transaction will read 51 as the value of x and x will have 52 as its value at the end of execution of these two transactions. However, since transactions are executing concurrently, the following execution sequence is also possible:

T_1: Read(x)
T_1: $x \leftarrow x+1$
T_2: Read(x)
T_1: Write(x)
T_2: $x \leftarrow x+1$
T_2: Write(x)
T_1: Commit
T_2: Commit

In this case, transaction T_2 reads 50 as the value of x. This is incorrect since T_2 reads x while its value is being changed from 50 to 51. Furthermore, the value of x is 51 at the end of execution of T_1 and T_2 since T_2's Write will overwrite T_1's Write. ♦

Ensuring isolation by not permitting incomplete results to be seen by other transactions, as the previous example shows, solves the *lost updates* problem. This type of isolation has been called *cursor stability*. In the example above, the second execution sequence resulted in the effects of T_1 being lost[3]. A second reason for isolation is *cascading aborts*. If a transaction permits others to see its incomplete results before committing and then decides to abort, any transaction that has read its incomplete values will have to abort as well. This chain can easily grow and impose considerable overhead on the DBMS.

It is possible to treat consistency levels discussed in the preceding section from the perspective of the isolation property (thus demonstrating the dependence between isolation and consistency). As we move up the hierarchy of consistency levels, there is more isolation among transactions. Degree 0 provides very little isolation other than preventing lost updates. However, since transactions commit write operations before the entire transaction is completed (and committed), if an abort occurs after some writes are committed to disk, the updates to data items that have been committed will need to be undone. Since at this level other transactions are allowed to read the dirty data, it may be necessary to abort them as well. Degree 2 consistency avoids cascading aborts. Degree 3 provides full isolation which forces one of the conflicting transactions to wait until the other one terminates. Such execution sequences are called *strict* and will be discussed further in the next chapter. It is obvious that the issue of isolation is directly related to database consistency and is therefore the topic of concurrency control.

[3] A more dramatic example may be to consider x to be your bank account and T_1 a transaction that executes as a result of your *depositing* money into your account. Assume that T_2 is a transaction that is executing as a result of your spouse *withdrawing* money from the account at another branch. If the same problem as described in Example 10.8 occurs and the results of T_1 are lost, you will be terribly unhappy. If, on the other hand, the results of T_2 are lost, the bank will be furious. A similar argument can be made for the reservation transaction example we have been considering.

ANSI, as part of the SQL2 (also known as SQL-92) standard specification, has defined a set of isolation levels [ANSI, 1992]. SQL isolation levels are defined on the basis of what ANSI call *phenomena* which are situations that can occur if proper isolation is not maintained. Three phenomena are specified:

Dirty Read: As defined earlier, dirty data refer to data items whose values have been modified by a transaction that has not yet committed. Consider the case where transaction T_1 modifies a data item value, which is then read by another transaction T_2 before T_1 performs a Commit or Abort. In case T_1 aborts, T_2 has read a value which never exists in the database.
A precise specification[4] of this phenomenon is as follows (where subscripts indicate the transaction identifiers)

$$\ldots, W_1(x), \ldots, R_2(x), \ldots, C_1(\text{or } A_1), \ldots, C_2(\text{or } A_2)$$

or

$$\ldots, W_1(x), \ldots, R_2(x), \ldots, C_2(\text{or } A_2), \ldots, C_1(\text{or } A_1)$$

Non-repeatable or Fuzzy read: Transaction T_1 reads a data item value. Another transaction T_2 then modifies or deletes that data item and commits. If T_1 then attempts to reread the data item, it either reads a different value or it can't find the data item at all; thus two reads within the same transaction T_1 return different results.
A precise specification of this phenomenon is as follows:

$$\ldots, R_1(x), \ldots, W_2(x), \ldots, C_1(\text{or } A_1), \ldots, C_2(\text{or } A_2)$$

or

$$\ldots, R_1(x), \ldots, W_2(x), \ldots, C_2(\text{or } A_2), \ldots, C_1(\text{or } A_1)$$

Phantom: The phantom condition that was defined earlier occurs when T_1 does a search with a predicate and T_2 inserts new tuples that satisfy the predicate. Again, the precise specification of this phenomenon is (where P is the search predicate)

$$\ldots, R_1(P), \ldots, W_2(y \text{ in } P), \ldots, C_1(\text{or } A_1), \ldots, C_2(\text{or } A_2)$$

or

$$\ldots, R_1(P), \ldots, W_2(y \text{ in } P), \ldots, C_2(\text{ or } A_2), \ldots, C_1(\text{or } A_1)$$

[4] The precise specifications of these phenomena are due to Berenson et al. [1995] and correspond to their *loose interpretations* which they indicate are the more appropriate interpretations.

Based on these phenomena, the isolation levels are defined as follows. The objective of defining multiple isolation levels is the same as defining multiple consistency levels.

Read uncommitted: For transactions operating at this level all three phenomena are possible.
Read committed: Fuzzy reads and phantoms are possible, but dirty reads are not.
Repeatable read: Only phantoms are possible.
Anomaly serializable: None of the phenomena are possible.

ANSI SQL standard uses the term "serializable" rather than "anomaly serializable." However, a serializable isolation level, as precisely defined in the next chapter, cannot be defined solely in terms of the three phenomena identified above; thus this isolation level is called "anomaly serializable" [Berenson et al., 1995]. The relationship between SQL isolation levels and the four levels of consistency defined in the previous section are also discussed in [Berenson et al., 1995].

One non-serializable isolation level that is commonly implemented in commercial products is *snapshot isolation* [Berenson et al., 1995]. Snapshot isolation provides repeatable reads, but not serializable isolation. Each transaction "sees" a snapshot of the database when it starts and its reads and writes are performed on this snapshot – thus the writes are not visible to other transactions and it does not see the writes of other transactions.

10.2.4 Durability

Durability refers to that property of transactions which ensures that once a transaction commits, its results are permanent and cannot be erased from the database. Therefore, the DBMS ensures that the results of a transaction will survive subsequent system failures. This is exactly why in Example 10.2 we insisted that the transaction commit before it informs the user of its successful completion. The durability property brings forth the issue of *database recovery*, that is, how to recover the database to a consistent state where all the committed actions are reflected. This issue is discussed further in Chapter 12.

10.3 Types of Transactions

A number of transaction models have been proposed in literature, each being appropriate for a class of applications. The fundamental problem of providing "ACID"ity usually remains, but the algorithms and techniques that are used to address them may be considerably different. In some cases, various aspects of ACID requirements are relaxed, removing some problems and adding new ones. In this section we provide an overview of some of the transaction models that have been proposed and then identify our focus in Chapters 11 and 12.

Transactions have been classified according to a number of criteria. One criterion is the duration of transactions. Accordingly, transactions may be classified as *online* or *batch* [Gray, 1987]. These two classes are also called *short-life* and *long-life* transactions, respectively. Online transactions are characterized by very short execution/response times (typically, on the order of a couple of seconds) and by access to a relatively small portion of the database. This class of transactions probably covers a large majority of current transaction applications. Examples include banking transactions and airline reservation transactions.

Batch transactions, on the other hand, take longer to execute (response time being measured in minutes, hours, or even days) and access a larger portion of the database. Typical applications that might require batch transactions are design databases, statistical applications, report generation, complex queries, and image processing. Along this dimension, one can also define a *conversational* transaction, which is executed by interacting with the user issuing it.

Another classification that has been proposed is with respect to the organization of the read and write actions. The examples that we have considered so far intermix their read and write actions without any specific ordering. We call this type of transactions *general*. If the transactions are restricted so that all the read actions are performed before any write action, the transaction is called a *two-step* transaction [Papadimitriou, 1979]. Similarly, if the transaction is restricted so that a data item has to be read before it can be updated (written), the corresponding class is called *restricted* (or *read-before-write*) [Stearns et al., 1976]. If a transaction is both two-step and restricted, it is called a *restricted two-step* transaction. Finally, there is the *action* model of transactions [Kung and Papadimitriou, 1979], which consists of the restricted class with the further restriction that each ⟨read, write⟩ pair be executed atomically. This classification is shown in Figure 10.3, where the generality increases upward.

Example 10.9. The following are some examples of the above-mentioned models. We omit the declaration and commit commands.
General:

$$T_1 : \{R(x), R(y), W(y), R(z), W(x), W(z), W(w), C\}$$

Two-step:

$$T_2 : \{R(x), R(y), R(z), W(x), W(z), W(y), W(w), C\}$$

Restricted:

$$T_3 : \{R(x), R(y), W(y), R(z), W(x), W(z), R(w), W(w), C\}$$

Note that T_3 has to read w before writing.
Two-step restricted:

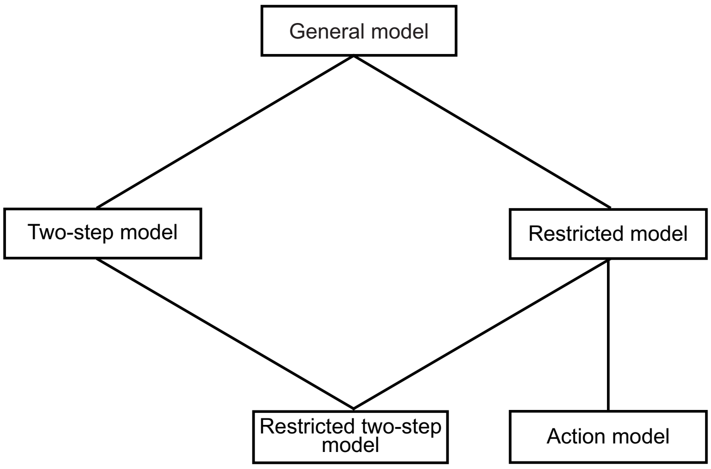

Fig. 10.3 Various Transaction Models (From: C.H. Papadimitriou and P.C. Kanellakis, ON CONCURRENCY CONTROL BY MULTIPLE VERSIONS. ACM Trans. Database Sys.; December 1984; 9(1): 89–99.)

$$T_4 : \{R(x), R(y), R(z), R(w), W(x), W(z), W(y), W(w), C\}$$

Action:

$$T_5 : \{[R(x), W(x)], [R(y), W(y)], [R(z), W(z)], [R(w), W(w)], C\}$$

Note that each pair of actions within square brackets is executed atomically. ♦

Transactions can also be classified according to their structure. We distinguish four broad categories in increasing complexity: *flat transactions*, *closed nested transactions* as in [Moss, 1985], and *open nested transactions* such as sagas [Garcia-Molina and Salem, 1987], and *workflow models* which, in some cases, are combinations of various nested forms. This classification is arguably the most dominant one and we will discuss it at some length.

10.3.1 Flat Transactions

Flat transactions have a single start point (**Begin_transaction**) and a single termination point (**End_transaction**). All our examples in this section are of this type. Most of the transaction management work in databases has concentrated on flat transactions. This model will also be our main focus in this book, even though we discuss management techniques for other transaction types, where appropriate.

10.3.2 Nested Transactions

An alternative transaction model is to permit a transaction to include other transactions with their own begin and commit points. Such transactions are called *nested* transactions. These transactions that are embedded in another one are usually called *subtransactions*.

Example 10.10. Let us extend the reservation transaction of Example 10.2. Most travel agents will make reservations for hotels and car rentals in addition to the flights. If one chooses to specify all of this as one transaction, the reservation transaction would have the following structure:

```
Begin_transaction Reservation
begin
   Begin_transaction Airline
                ...
   end. {Airline}
   Begin_transaction Hotel
                ...
   end. {Hotel}
   Begin_transaction Car
                ...
   end. {Car}
end.
```

♦

Nested transactions have received considerable interest as a more generalized transaction concept. The level of nesting is generally open, allowing subtransactions themselves to have nested transactions. This generality is necessary to support application areas where transactions are more complex than in traditional data processing.

In this taxonomy, we differentiate between *closed* and *open* nesting because of their termination characteristics. Closed nested transactions [Moss, 1985] commit in a bottom-up fashion through the root. Thus, a nested subtransaction begins *after* its parent and finishes *before* it, and the commitment of the subtransactions is conditional upon the commitment of the parent. The semantics of these transactions enforce atomicity at the top-most level. Open nesting relaxes the top-level atomicity restriction of closed nested transactions. Therefore, an open nested transaction allows its partial results to be observed outside the transaction. Sagas [Garcia-Molina and Salem, 1987; Garcia-Molina et al., 1990] and split transactions [Pu, 1988] are examples of open nesting.

A saga is a “sequence of transactions that can be interleaved with other transactions” [Garcia-Molina and Salem, 1987]. The DBMS guarantees that either all the transactions in a saga are successfully completed or *compensating transactions* [Garcia-Molina, 1983; Korth et al., 1990] are run to recover from a partial execution. A compensating transaction effectively does the inverse of the transaction that it is associated with. For example, if the transaction adds $100 to a bank account, its compensating transaction deducts $100 from the same bank account. If a transaction is viewed as a function that maps the old database state to a new database state, its compensating transaction is the inverse of that function.

Two properties of sagas are: (1) only two levels of nesting are allowed, and (2) at the outer level, the system does not support full atomicity. Therefore, a saga differs from a closed nested transaction in that its level structure is more restricted (only 2) and that it is open (the partial results of component transactions or sub-sagas are visible to the outside). Furthermore, the transactions that make up a saga have to be executed sequentially.

The saga concept is extended and placed within a more general model that deals with long-lived transactions and with activities that consist of multiple steps [Garcia-Molina et al., 1990] . The fundamental concept of the model is that of a module that captures code segments each of which accomplishes a given task and access a database in the process. The modules are modeled (at some level) as sub-sagas that communicate with each other via messages over ports. The transactions that make up a saga can be executed in parallel. The model is multi-layer where each subsequent layer adds a level of abstraction.

The advantages of nested transactions are the following. First, they provide a higher-level of concurrency among transactions. Since a transaction consists of a number of other transactions, more concurrency is possible within a single transaction. For example, if the reservation transaction of Example 10.10 is implemented as a flat transaction, it may not be possible to access records about a specific flight concurrently. In other words, if one travel agent issues the reservation transaction for a given flight, any concurrent transaction that wishes to access the same flight data will have to wait until the termination of the first, which includes the hotel and car reservation activities in addition to flight reservation. However, a nested implementation will permit the second transaction to access the flight data as soon as the Airline subtransaction of the first reservation transaction is completed. In other words, it may be possible to perform a finer level of synchronization among concurrent transactions.

A second argument in favor of nested transactions is related to recovery. It is possible to recover independently from failures of each subtransaction. This limits the damage to a smaller part of the transaction, making it less costly to recover. In a flat transaction, if any operation fails, the entire transaction has to be aborted and restarted, whereas in a nested transaction, if an operation fails, only the subtransaction containing that operation needs to be aborted and restarted.

Finally, it is possible to create new transactions from existing ones simply by inserting the old one inside the new one as a subtransaction.

10.3.3 Workflows

Flat transactions model relatively simple and short activities very well. However, they are less appropriate for modeling longer and more elaborate activities.That is the reason for the development of the various nested transaction models discussed above. It has been argued that these extensions are not sufficiently powerful to model business activities: "after several decades of data processing, we have learned that we have not won the battle of modeling and automating complex enterprises" [Medina-Mora et al., 1993]. To meet these needs, more complex transaction models which are combinations of open and nested transactions have been proposed. There are well-justified arguments for not calling these transactions, since they hardly follow any of the ACID properties; a more appropriate name that has been proposed is a *workflow* [Dogac et al., 1998b; Georgakopoulos et al., 1995].

The term "workflow," unfortunately, does not have a clear and uniformly accepted meaning. A working definition is that a workflow is "a collection of *tasks* organized to accomplish some business process." [Georgakopoulos et al., 1995]. This definition, however, leaves a lot undefined. This is perhaps unavoidable given the very different contexts where this term is used. Three types of workflows are identified [Georgakopoulos et al., 1995]:

1. *Human-oriented workflows*, which involve humans in performing the tasks. The system support is provided to facilitate collaboration and coordination among humans, but it is the humans themselves who are ultimately responsible for the consistency of the actions.
2. *System-oriented workflows* are those that consist of computation-intensive and specialized tasks that can be executed by a computer. The system support in this case is substantial and involves concurrency control and recovery, automatic task execution, notification, etc.
3. *Transactional workflows* range in between human-oriented and system-oriented workflows and borrow characteristics from both. They involve "coordinated execution of multiple tasks that (a) may involve humans, (b) require access to HAD [heterogeneous, autonomous, and/or distributed] systems, and (c) support selective use of transactional properties [i.e., ACID properties] for individual tasks or entire workflows." [Georgakopoulos et al., 1995].
   Among the features of transactional workflows, the selective use of transactional properties is particularly important as it characterizes possible relaxations of ACID properties.

In this book, our primary interest is with transactional workflows. There have been many transactional workflow proposals [Elmagarmid et al., 1990; Nodine and Zdonik, 1990; Buchmann et al., 1982; Dayal et al., 1991; Hsu, 1993], and they differ in a number of ways. The common point among them is that a workflow is defined as an *activity* consisting of a set of tasks with well-defined precedence relationship among them.

Example 10.11. Let us further extend the reservation transaction of Example 10.3. The entire reservation activity consists of the following taks and involves the following data:

- Customer request is obtained (task T_1) and Customer Database is accessed to obtain customer information, preferences, etc.;
- Airline reservation is performed (T_2) by accessing the Flight Database;
- Hotel reservation is performed (T_3), which may involve sending a message to the hotel involved;
- Auto reservation is performed (T_4), which may also involve communication with the car rental company;
- Bill is generated (T_5) and the billing info is recorded in the billing database.

Figure 10.4 depicts this workflow where there is a serial dependency of T_2 on T_1, and T_3, T_4 on T_2; however, T_3 and T_4 (hotel and car reservations) are performed in parallel and T_5 waits until their completion. ♦

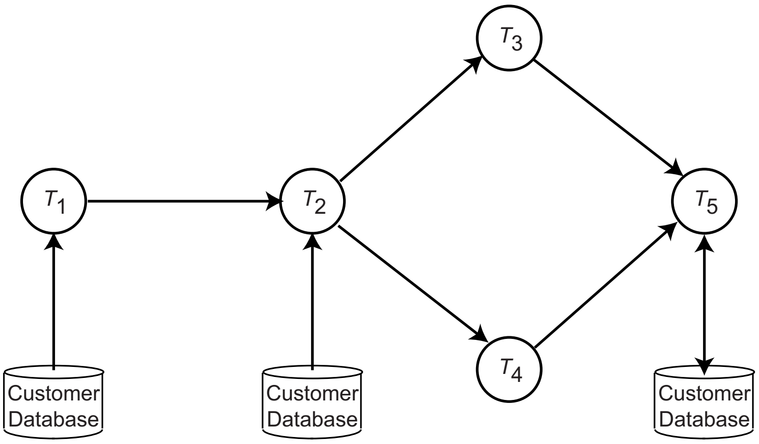

Fig. 10.4 Example Workflow

A number of workflow models go beyond this basic model by both defining more precisely what tasks can be and by allocating different relationships among the tasks. In the following, we define one model that is similar to the models of Buchmann et al. [1982] and Dayal et al. [1991].

A workflow is modeled as an *activity* with open nesting semantics in that it permits partial results to be visible outside the activity boundaries. Thus, tasks that make up the activity are allowed to commit individually. Tasks may be other activities (with the same open transaction semantics) or closed nested transactions that make their results visible to the entire system when they commit. Even though an activity can have both other activities and closed nested transactions as its component, a closed nested transaction task can only be composed of other closed nested transactions (i.e., once closed nesting semantics begins, it is maintained for all components).

An activity commits when its components are ready to commit. However, the components commit individually, without waiting for the root activity to commit.

This raises problems in dealing with aborts since when an activity aborts, all of its components should be aborted. The problem is dealing with the components that have already committed. Therefore, compensating transactions are defined for the components of an activity. Thus, if a component has already committed when an activity aborts, the corresponding compensating transaction is executed to "undo" its effects.

Some components of an activity may be marked as *vital*. When a vital component aborts, its parent must also abort. If a non-vital component of a workflow model aborts, it may continue executing. A workflow, on the other hand, always aborts when one of its components aborts. For example, in the reservation workflow of Example 10.11, T_2 (airline reservation) and T_3 (hotel reservation) may be declared as vital so that if an airline reservation or a hotel reservation cannot be made, the workflow aborts and the entire trip is canceled. However, if a car reservation cannot be committed, the workflow can still successfully terminate.

It is possible to define *contingency tasks* that are invoked if their counterparts fail. For example, in the Reservation example presented earlier, one can specify that the contingency to making a reservation at Hilton is to make a reservation at Sheraton. Thus, if the hotel reservation component for Hilton fails, the Sheraton alternative is tried rather than aborting the task and the entire workflow.

10.4 Architecture Revisited

With the introduction of the transaction concept, we need to revisit the architectural model introduced in Chapter 1. We do not need to revise the model but simply need to expand the role of the distributed execution monitor.

The distributed execution monitor consists of two modules: a *transaction manager* (TM) and a *scheduler* (SC). The transaction manager is responsible for coordinating the execution of the database operations on behalf of an application. The scheduler, on the other hand, is responsible for the implementation of a specific concurrency control algorithm for synchronizing access to the database.

A third component that participates in the management of distributed transactions is the local recovery managers (LRM) that exist at each site. Their function is to implement the local procedures by which the local database can be recovered to a consistent state following a failure.

Each transaction originates at one site, which we will call its *originating site*. The execution of the database operations of a transaction is coordinated by the TM at that transaction's originating site.

The transaction managers implement an interface for the application programs which consists of five commands: begin_transaction, read, write, commit, and abort. The processing of each of these commands in a non-replicated distributed DBMS is discussed below at an abstract level. For simplicity, we ignore the scheduling of concurrent transactions as well as the details of how data is physically retrieved by the data processor. These assumptions permit us to concentrate on the interface to the TM. The details are presented in the Chapters 11 and 12, while the execution of these commands in a replicated distributed database is discussed in Chapter 13.

1. *Begin_transaction*. This is an indicator to the TM that a new transaction is starting. The TM does some bookkeeping, such as recording the transaction's name, the originating application, and so on, in coordination with the data processor.
2. *Read*. If the data item to be read is stored locally, its value is read and returned to the transaction. Otherwise, the TM finds where the data item is stored and requests its value to be returned (after appropriate concurrency control measures are taken).
3. *Write*. If the data item is stored locally, its value is updated (in coordination with the data processor). Otherwise, the TM finds where the data item is located and requests the update to be carried out at that site after appropriate concurrency control measures are taken).
4. *Commit*. The TM coordinates the sites involved in updating data items on behalf of this transaction so that the updates are made permanent at every site.
5. *Abort*. The TM makes sure that no effects of the transaction are reflected in any of the databases at the sites where it updated data items.

In providing these services, a TM can communicate with SCs and data processors at the same or at different sites. This arrangement is depicted in Figure 10.5.

As we indicated in Chapter 1, the architectural model that we have described is only an abstraction that serves a pedagogical purpose. It enables the separation of many of the transaction management issues and their independent and isolated discussion. In Chapter 11 we focus on the interface between a TM and an SC and between an SC and a data processor, in addition to the scheduling algorithms. In Chapter 12 we consider the execution strategies for the commit and abort commands in a distributed environment, in addition to the recovery algorithms that need to be implemented for the recovery manager. In Chapter 13, we extend this discussion to the case of replicated databases. We should point out that the computational model that we described here is not unique. Other models have been proposed such as, for example, using a private workspace for each transaction.

10.5 Conclusion

In this chapter we introduced the concept of a transaction as a unit of consistent and reliable access to the database. The properties of transactions indicate that they are larger atomic units of execution which transform one consistent database to another consistent database. The properties of transactions also indicate what the requirements for managing them are, which is the topic of the next two chapters. Consistency requires a definition of integrity enforcement (which we did in Chapter

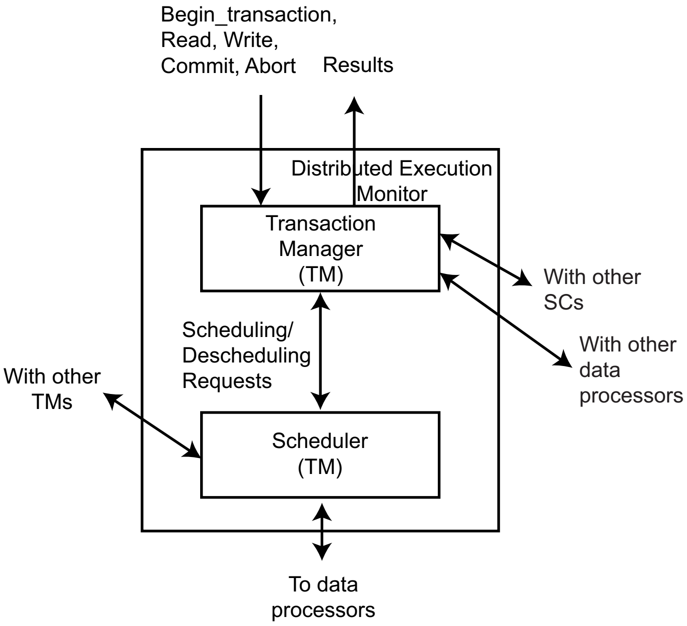

Fig. 10.5 Detailed Model of the Distributed Execution Monitor

5), as well as concurrency control algorithms (which is the topic of Chapter 11). Concurrency control also deals with the issue of isolation. Durability and atomicity properties of transactions require a discussion of reliability, which we cover in Chapter 12. Specifically, durability is supported by various commit protocols and commit management, whereas atomicity requires the development of appropriate recovery protocols.

10.6 Bibliographic Notes

Transaction management has been the topic of considerable study since DBMSs have become a significant research area. There are two excellent books on the subject: [Gray and Reuter, 1993] and [Weikum and Vossen, 2001]. An excellent companion to these is [Bernstein and Newcomer, 1997] which provides an in-depth discussion of transaction processing principles. It also gives a view of transaction processing and transaction monitors which is more general than the database-centric view that we provide in this book. A good collection of papers that focus on the concurrency control and reliability aspects of distributed systems is [Bhargava, 1987]. Two books focus on the performance of concurrency control mechanisms with a focus on centralized systems [Kumar, 1996; Thomasian, 1996]. Distributed concurrency control is the topic of [Cellary et al., 1988].

Advanced transaction models are discussed and various examples are given in [Elmagarmid, 1992]. Nested transactions are also covered in [Lynch et al., 1993]. A good introduction to workflow systems is [Georgakopoulos et al., 1995]. The same topic is covered in detail in [Dogac et al., 1998b].

A very important work is a set of notes on database operating systems by Gray [1979]. These notes contain valuable information on transaction management, among other things.

The discussion concerning transaction classification in Section 10.3 comes from a number of sources. Part of it is from [Farrag, 1986]. The structure discussion is from [Özsu, 1994] and [Buchmann et al., 1982], where the authors combine transaction structure with the structure of the objects that these transactions operate upon to develop a more complete classification.

There are numerous papers dealing with various transaction management issues. The ones referred to in this chapter are those that deal with the concept of a transaction. More detailed references on their management are left to Chapters 11 and 12.