Appendix C
Transaction Processing Fundamentals

In Appendix B our focus was on retrieve-only (or read-only) queries that read data from a 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. That is the focus of this Appendix that focuses on the concept of a *transaction* as the basic unit for consistent and reliable computation.

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. At this point, we are using 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 3). 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 C.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

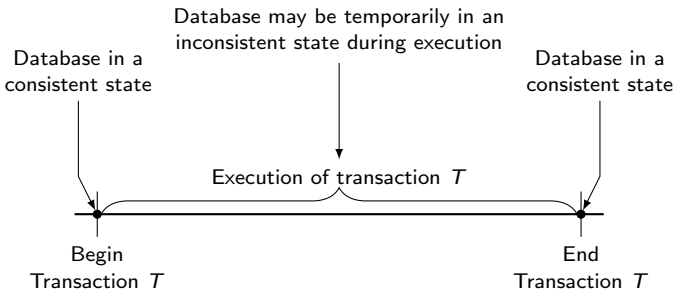

Fig. C.1: A Transaction Model

are a number of user requests that are concurrently accessing (reading or updating) the database.

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.

C.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 C.1. Consider the following SQL query for increasing by 10% the budget of the CAD/CAM project:

```
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 C.2. In our discussion of transaction management concepts, we will use an airline reservation system example. 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, FDATE, SRC, DEST, STSOLD, CAP)

CUST(CNAME, ADDR, BAL)

FC(FNO, FDATE, CNAME, SPECIAL)

The definition of the attributes in this database schema are as follows: FNO is the flight number, FDATE 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, fdate, customer_name)
   EXEC SQL UPDATE FLIGHT
        SET STSOLD = STSOLD + 1
        WHERE FNO = flight_no
        AND FDATE = fdate;
   EXEC SQL INSERT
        INTO FC(FNO,FDATE,CNAME,SPECIAL)
        VALUES (flight_no, fdate,customer_name, null};
   output("reservation completed")
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. ♦

C.1.1 Termination Conditions of Transactions

The reservation transaction of Example C.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 Section C.6. 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 later. 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 Section C.6.

Example C.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, fdate, customer_name)
   EXEC SQL SELECT STSOLD, CAP
        INTO temp1, temp2
        FROM FLIGHT
        WHERE FNO = flight_no
        AND FDATE = fdate;
   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 FDATE = fdate;
       EXEC SQL INSERT
          INTO FC(FNO,FDATE,CNAME,SPECIAL)
          VALUES (flight_no, fdate,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 Sections C.2.4 and C.6.

C.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 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 C.4. Considering the reservation transaction as specified in Example C.3 and the insert to be a number of write operations, the above-mentioned sets are defined as follows:

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

Note that it may be appropriate to include FLIGHT.FNO and FLIGHT.FDATE 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

[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 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 in many textbooks.

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.

C.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

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

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 C.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 $O_i \prec O_j$. ♦

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 C.6. The reservation transaction developed in Example C.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:

$$\Sigma = \{R(\texttt{STSOLD}), R(\texttt{CAP}), A\}$$
$$\prec = \{(O_1, A), (O_2, A)\}$$

and the latter can be specified as

$$\Sigma = \{R(\texttt{STSOLD}), R(\texttt{CAP}), W(\texttt{STSOLD}), W(\texttt{FNO}), W(\texttt{FDATE}), W(\texttt{CNAME}), W(\texttt{SPECIAL}), C\}$$
$$\prec = \{(O_1, O_3), (O_2, O_3), (O_1, O_4), (O_1, O_5), (O_1, O_6), (O_1, O_7), (O_2, O_4), (O_2, O_5), (O_2, O_6), (O_2, O_7), (O_1, C), (O_2, C), (O_3, C), (O_4, C), (O_5, C), (O_6, C), (O_7, C)\}$$

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

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 (Section C.5) and in arguing about their correctness by means of graph-theoretic tools.

Example C.7. The transaction discussed in Example C.5 can be represented as a DAG as depicted in Figure C.2. Note that we do not draw the arcs that are implied by transitivity even though we indicate them as elements of $\prec$. ♦

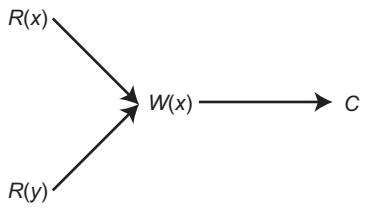

Fig. C.2: DAG Representation of a Transaction

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 C.5 as follows:

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

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

C.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.

C.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 C.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 Section C.6.

C.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 that we don't discuss any further (for discussion of the issue within the context of distributed DBMS, see Chapter 3). 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 Section C.5.

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.

C.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 C.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.

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

SQL defines a number of isolation levels based on *phenomena* which are situations that can occur if proper isolation is not maintained [ANSI, 1992]. 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

[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 C.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.

[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.

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

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

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.

SQL standard uses the term "serializable" rather than "anomaly serializable." However, a serializable isolation level, as precisely defined in Section C.5, 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.

C.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 C.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 Section C.6.

C.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 Sections C.5 and C.6.

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 C.3, where the generality increases upward.

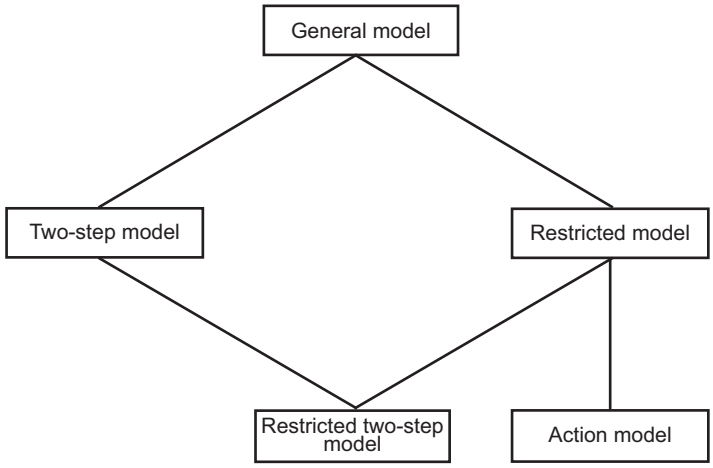

Fig. C.3: Various Transaction Models (From: C.H. Papadimitriou and P.C. Kanellakis, ON CONCURRENCY CONTROL BY MULTIPLE VERSIONS. ACM Trans. Database Sys.; December 1984; 9(1): 89–99.)

Example C.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:

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

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.

C.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.

C.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 C.10. Let us extend the reservation transaction of Example C.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 C.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.

C.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., 1998; 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., 1992; 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 C.11. Let us further extend the reservation transaction of Example C.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 C.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. ♦

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. [1992] 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).

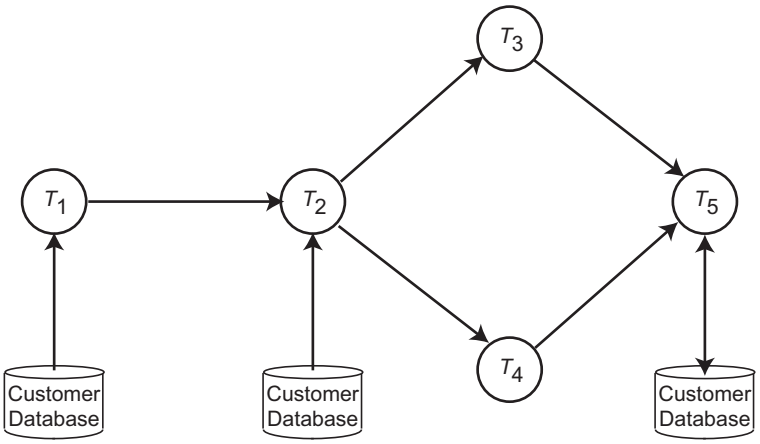

Fig. C.4: Example Workflow

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 C.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.

C.4 Transaction Processing Architecture

We now discuss an abstract architecture for transaction processing. The architecture is abstract, because it does not necessarily reflect any actual system implementation, but focuses on the *functions* that need to exist in any actual system. There are different ways these functions can be implemented.

The abstract architecture is shown in Figure C.5. The transaction manager (TM) is the interface for applications to the transaction functionality and is responsible for coordinating the execution of the database operations on behalf of an application. The scheduler (SC), 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 recovery manager (RM), which implements the procedures by which database consistency can be maintained even in the face of failures and the database can be recovered to a consistent state following a failure.

All access to the database is via the database buffer manager that coordinates the movement of data between secondary storage (which stores the *stable database*) and main memory buffers (where *volatile database* is maintained).

The transaction manager implements 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 DBMS is discussed below with the details deferred to Sections C.5 and C.6.

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, through a `begin_transaction` record in the log[5].
2. *Read*. The data item's value is read and returned to the transaction.
3. *Write*. The data item's value is modified in the database. We use *Write* to model Update, Insert or Delete on the database.
4. *Commit*. The TM coordinates that the updates are made permanent.
5. *Abort*. The TM makes sure that no effects of the transaction are reflected in the database.

The database is typically stored permanently on secondary storage, which in this context is called the *stable storage* [Lampson and Sturgis, 1976]. The stability of this storage medium is due to its robustness to failures. A stable storage device would experience considerably less-frequent failures than would a non-stable storage device. We call the version of the database that is kept on stable storage the *stable database*. The unit of storage and access of the stable database is typically a *page*.

---

[5] For the time being, we ignore the logs and logging in Figure C.5; these will be discussed in Section C.6.2.

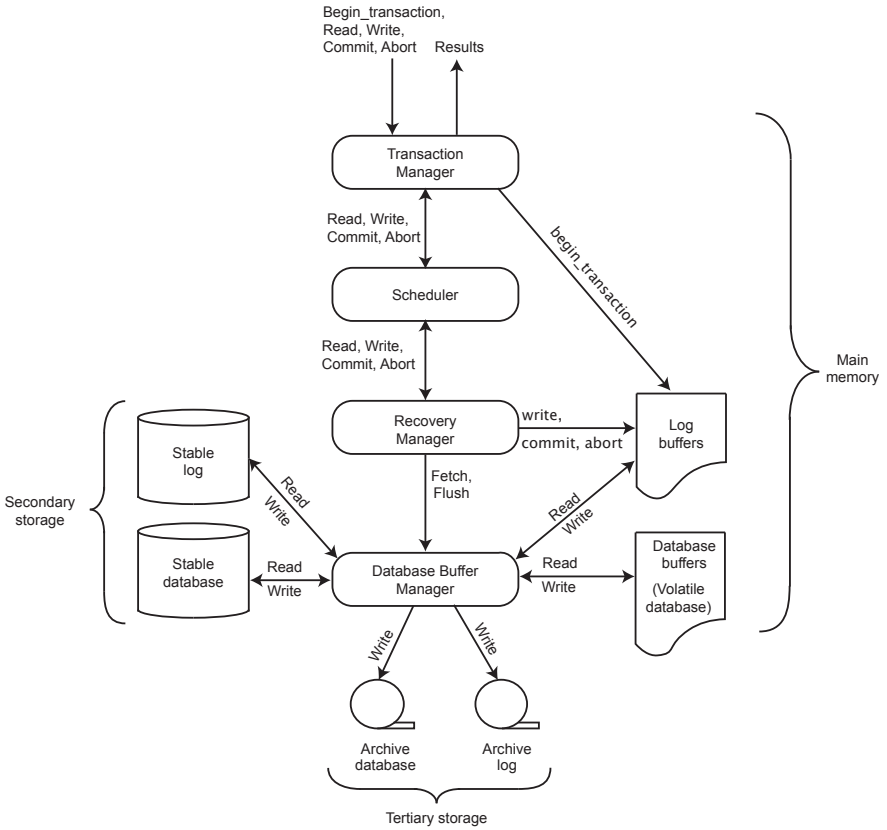

Fig. C.5: Abstract Transaction Processign Architecture

The database buffer manager keeps some of the recently accessed data in main memory buffers where the applications can access them. Typically, the buffer is divided into pages that are of the same size as the stable database pages. The part of the database that is in the database buffer is called the *volatile database*. It is important to note that the RM executes the operations on behalf of a transaction only on the volatile database, which, at a later time, is written back to the stable database. The movement of the data between the stable database and the volatile database is the responsibility of the buffer manager and is discussed in Section C.6.2.

C.5 Concurrency Control

Concurrency control deals with the isolation and consistency properties of transactions. The concurrency control mechanism of a DBMS ensures that the consistency of the

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

In this section, we make a major assumption: the DBMS and the underlying system is fully reliable and does not experience any failures (of hardware or software). We make this (unrealistic) assumption to isolate the issues related to the management of concurrency from those related to the operation of a reliable system. In Section C.6, we discuss how the algorithms that are presented in this section need to be enhanced to operate in an unreliable environment.

C.5.1 Serializability Theory

In Section C.1.3 we discussed the issue of isolating transactions from one another in terms of their effects on the database. We also pointed out that if the concurrent execution of transactions leaves the database in a state that can be achieved by their serial execution in some order, problems such as lost updates will be resolved. This is exactly the point of the serializability argument. The remainder of this section addresses serializability issues more formally.

A *history* R (also called a *schedule*) is defined over a set of transactions $T = \{T_1, T_2, \ldots, T_n\}$ and specifies an interleaved order of execution of these transactions' operations. Based on the definition of a transaction introduced in Section C.1, the history can be specified as a partial order over T. We need a few preliminaries, though, before we present the formal definition.

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

We first define a *complete history*, which defines the execution order of all operations in its domain. We will then define a history as a prefix of a complete history.

Formally, a complete history H_T^c defined over a set of transactions $T = \{T_1, T_2, \ldots, T_n\}$ is a partial order $H_T^c = \{\Sigma_T, \prec_H\}$ where

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

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

Example C.12. Consider the two transactions from Example C.8, which were as follows:

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

A possible complete history H_T^c over $T = \{T_1, T_2\}$ is the partial order $H_T^c = \{\Sigma_T, \prec_T\}$ where

$$\Sigma_1 = \{R_1(x), W_1(x), C_1\}$$
$$\Sigma_2 = \{R_2(x), W_2(x), C_2\}$$

Thus

$$\Sigma_T = \Sigma_1 \cup \Sigma_2 = \{R_1(x), W_1(x), C_1, R_2(x), W_2(x), C_2\}$$

and

$$\prec_H = \{(R_1, R_2), (R_1, W_1), (R_1, C_1), (R_1, W_2), (R_1, C_2), (R_2, W_1), (R_2, C_1), (R_2, W_2), (R_2, C_2), (W_1, C_1), (W_1, W_2), (W_1, C_2), (C_1, W_2), (C_1, C_2), (W_2, C_2)\}$$

which can be specified as a DAG as depicted in Figure C.6. Note that consistent with our earlier adopted convention (see Example C.7), we do not draw the arcs that are implied by transitivity [e.g., (R_1, C_1)].

It is quite common to specify a history as a listing of the operations in Σ_T, where their execution order is relative to their order in this list. Thus H_T^c can be specified as

$$H_T^c = \{R_1(x), R_2(x), W_1(x), C_1, W_2(x), C_2\}$$

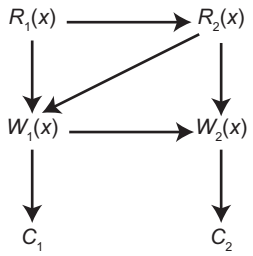

Fig. C.6: DAG Representation of a Complete History

♦

A history is defined as a prefix of a complete history. A prefix of a partial order can be defined as follows. Given a partial order $P = \{\Sigma, \prec\}$, $P' = \{\Sigma', \prec'\}$ is a *prefix* of P if

1. $\Sigma' \subseteq \Sigma$;
2. $\forall e_i \in \Sigma', e_1 \prec' e_2$ if and only if $e_1 \prec e_2$; and
3. $\forall e_i \in \Sigma'$, if $\exists e_j \in \Sigma$ and $e_j \prec e_i$, then $e_j \in \Sigma'$.

The first two conditions define P' as a *restriction* of P on domain Σ', whereby the ordering relations in P are maintained in P'. The last condition indicates that for any element of Σ', all its predecessors in Σ have to be included in Σ' as well.

What does this definition of a history as a prefix of a partial order provide for us? The answer is simply that we can now deal with incomplete histories. This is useful for a number of reasons. From the perspective of the serializability theory, we deal only with conflicting operations of transactions rather than with all operations. Furthermore, and perhaps more important, when we introduce failures, we need to be able to deal with incomplete histories, which is what a prefix enables us to do.

The history discussed in Example C.12 is complete. It needs to be complete in order to talk about the execution order of these two transactions' operations. The following example demonstrates a history that is not complete.

Example C.13. Consider the following three transactions:

T_1: Read(x)	T_2: Write(x)	T_3: Read(x)
Write(x)	Write(y)	Read(y)
Commit	Read(z)	Read(z)
	Commit	Commit

A complete history H^c for these transactions is given in Figure C.7, and a history H (as a prefix of H^c) is depicted in Figure C.8. ♦

If in a complete history H, the operations of various transactions are not interleaved (i.e., the operations of each transaction occur consecutively), the history is said to be

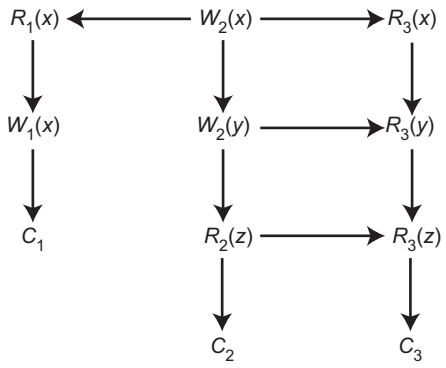

Fig. C.7: A Complete History

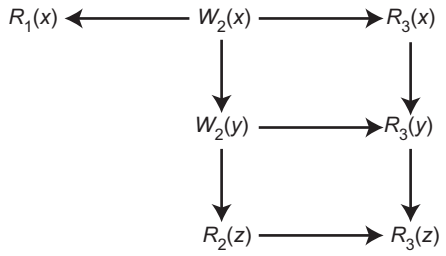

Fig. C.8: Prefix of Complete History in Figure C.7

serial. As we indicated before, the serial execution of a set of transactions maintains the consistency of the database. This follows naturally from the consistency property of transactions: each transaction, when executed alone on a consistent database, will produce a consistent database.

Example C.14. Consider the three transactions of Example C.13. The following history is serial since all the operations of T_2 are executed before all the operations of T_1 and all operations of T_1 are executed before all operations of T_3[6].

$$H = \{\underbrace{W_2(x), W_2(y), R_2(z)}_{T_2}, \underbrace{R_1(x), W_1(x)}_{T_1}, \underbrace{R_3(x), R_3(y), R_3(z)}_{T_3}\}$$

One common way to denote this precedence relationship between transaction executions is $T_2 \rightarrow T_1 \rightarrow T_3$ rather than the more formal $T_2 \prec_H T_1 \prec_H T_3$. ♦

Based on the precedence relationship introduced by the partial order, it is possible to discuss the equivalence of histories with respect to their effects on the database. Intuitively, two histories H_1 and H_2, defined over the same set of transactions T, are

[6] From now on we will generally omit the Commit operation from histories.

equivalent if they have the same effect on the database. More formally, two histories, H_1 and H_2, defined over the same set of transactions T, are said to be *equivalent* if for each pair of conflicting operations O_{ij} and O_{kl} $(i \neq k)$, whenever $O_{ij} \prec_{H_1} O_{kl}$, then $O_{ij} \prec_{H_2} O_{kl}$. This is called *conflict equivalence* since it defines equivalence of two histories in terms of the relative order of execution of the conflicting operations in those histories. Here, for the sake of simplicity, we assume that T does not include any aborted transaction. Otherwise, the definition needs to be modified to specify only those conflicting operations that belong to unaborted transactions.

Example C.15. Again consider the three transactions given in Example C.13. The following history H' defined over them is conflict equivalent to H given in Example C.14:

$$H' = \{W_2(x), R_1(x), W_1(x), R_3(x), W_2(y), R_3(y), R_2(z), R_3(z)\}$$

♦

We are now ready to define serializability more precisely. A history H is said to be *serializable* if and only if it is conflict equivalent to a serial history. Note that serializability roughly corresponds to degree 3 consistency, which we defined in Section C.2.2. Serializability so defined is also known as *conflict-based serializability* since it is defined according to conflict equivalence.

Example C.16. History H' in Example C.15 is serializable since it is equivalent to the serial history H of Example C.14. Also note that the problem with the uncontrolled execution of transactions T_1 and T_2 in Example C.8 was that they could generate an unserializable history. ♦

Now that we have formally defined serializability, we can indicate that the primary function of a concurrency controller is to generate a serializable history for the execution of pending transactions. The issue, then, is to devise algorithms that are guaranteed to generate only serializable histories.

Serializability theory extends in a straightforward manner to the non-replicated (or partitioned) distributed databases. The history of transaction execution at each site is called a *local history*. If the database is not replicated and each local history is serializable, their union (called the *global history*) is also serializable as long as local serialization orders are identical.

Example C.17. We will give a very simple example to demonstrate the point. Consider two bank accounts, x (stored at Site 1) and y (stored at Site 2), and the following two transactions where T_1 transfers \$100 from x to y, while T_2 simply reads the balances of x and y:

T_1: Read(x)	T_2: Read(x)
$x \leftarrow x - 100$	Read(y)
Write(x)	Commit
Read(y)	
$y \leftarrow y + 100$	
Write(y)	
Commit	

Obviously, both of these transactions need to run at both sites. Consider the following two histories that may be generated locally at the two sites (H_i is the history at Site i):

$$H_1 = \{R_1(x), W_1(x), R_2(x)\}$$
$$H_2 = \{R_1(y), W_1(y), R_2(y)\}$$

Both of these histories are serializable; indeed, they are serial. Therefore, each represents a correct execution order. Furthermore, the serialization order for both are the same $T_1 \rightarrow T_2$. Therefore, the global history that is obtained is also serializable with the serialization order $T_1 \rightarrow T_2$.

However, if the histories generated at the two sites are as follows, there is a problem:

$$H_1^{'} = \{R_1(x), W_1(x), R_2(x)\}$$
$$H_2^{'} = \{R_2(y), R_1(y), W_1(y)\}$$

Although each local history is still serializable, the serialization orders are different: $H_1^{'}$ serializes T_1 before T_2 while $H_2^{'}$ serializes T_2 before T_1. Therefore, there can be no global history that is serializable. ♦

A weaker version of serializability that has gained importance in recent years is *snapshot isolation* [Berenson et al., 1995] that is now provided as a standard consistency criterion in a number of commercial systems. Snapshot isolation allows read transactions (queries) to read stale data by allowing them to read a snapshot of the database that reflects the committed data at the time the read transaction starts. Consequently, the reads are never blocked by writes, even though they may read old data that may be dirtied by other transactions that were still running when the snapshot was taken. Hence, the resulting histories are not serializable, but this is accepted as a reasonable tradeoff between a lower level of isolation and better performance.

C.5.2 Taxonomy of Concurrency Control Mechanisms

There are a number of ways that the concurrency control approaches can be classified. One obvious classification criterion is the mode of database distribution. Some

algorithms that have been proposed require a fully replicated database, while others can operate on partially replicated or partitioned databases. The concurrency control algorithms may also be classified according to network topology, such as those requiring a communication subnet with broadcasting capability or those working in a star-type network or a circularly connected network.

The most common classification criterion, however, is the synchronization primitive. The corresponding breakdown of the concurrency control algorithms results in two classes [Bernstein and Goodman, 1981]: those algorithms that are based on mutually exclusive access to shared data (locking), and those that attempt to order the execution of the transactions according to a set of rules (protocols). However, these primitives may be used in algorithms with two different viewpoints: the pessimistic view that many transactions will conflict with each other, or the optimistic view that not too many transactions will conflict with one another.

We will thus group the concurrency control mechanisms into two broad classes: pessimistic concurrency control methods and optimistic concurrency control methods. *Pessimistic* algorithms synchronize the concurrent execution of transactions early in their execution life cycle, whereas *optimistic* algorithms delay the synchronization of transactions until their termination. The pessimistic group consists of *locking-based* algorithms, *ordering* (or *transaction ordering*) *based* algorithms, and *hybrid* algorithms. The optimistic group can, similarly, be classified as locking-based or timestamp ordering-based. This classification is depicted in Figure C.9.

In the *locking-based* approach, the synchronization of transactions is achieved by employing physical or logical locks on some portion or granule of the database. The size of these portions (usually called *locking granularity*) is an important issue. However, for the time being, we will ignore it and refer to the chosen granule as a *lock unit*. This class is subdivided further according to where the lock management activities are performed: *centralized* and *decentralized* (or *distributed*) *locking*.

The *timestamp ordering* (TO) class involves organizing the execution order of transactions so that they maintain transaction consistency. This ordering is maintained by assigning timestamps to both the transactions and the data items that are stored in the database. These algorithms can be *basic TO*, *multiversion TO*, or *conservative TO*.

In some locking-based algorithms, timestamps are also used. This is done primarily to improve efficiency and the level of concurrency. We call these *hybrid* algorithms. We will not discuss these algorithms, since they have not been implemented in any commercial or research prototype distributed DBMS. The rules for integrating locking and timestamp ordering protocols are discussed by Bernstein and Goodman [1981].

C.5.3 Locking-Based Concurrency Control

The main idea of locking-based concurrency control is to ensure that a data item that is shared by conflicting operations is accessed by one operation at a time. This

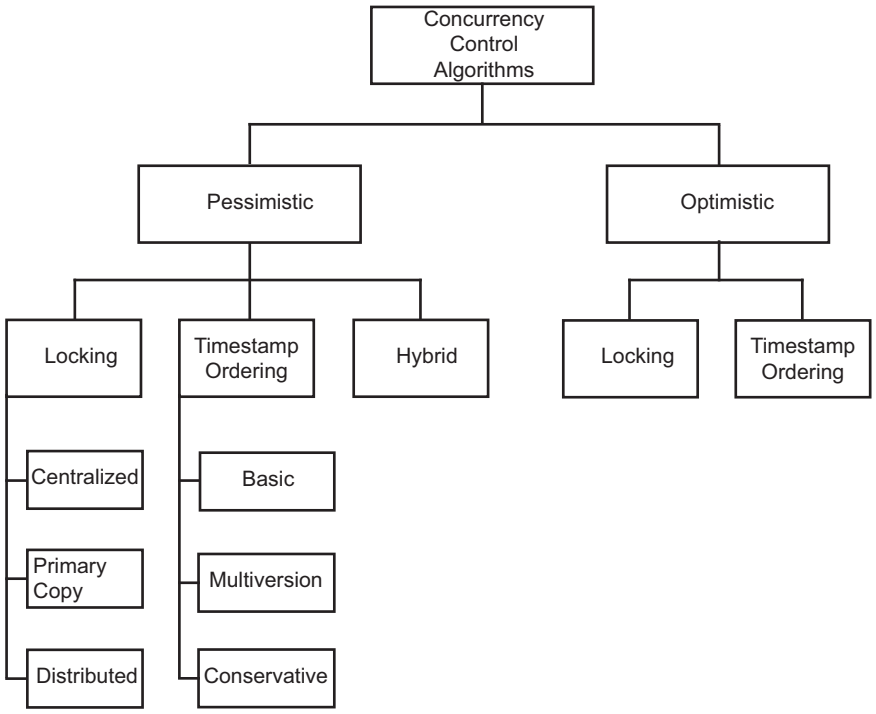

Fig. C.9: Classification of Concurrency Control Algorithms

is accomplished by associating a "lock" with each lock unit. This lock is set by a transaction before it is accessed and is reset at the end of its use. Obviously a lock unit cannot be accessed by an operation if it is already locked by another. Thus a lock request by a transaction is granted only if the associated lock is not being held by any other transaction.

Since we are concerned with synchronizing the conflicting operations of conflicting transactions, there are two types of locks (commonly called *lock modes*) associated with each lock unit: *read lock* (*rl*) and *write lock* (*wl*). A transaction T_i that wants to read a data item contained in lock unit x obtains a read lock on x [denoted $rl_i(x)$]. The same happens for write operations. Two lock modes are *compatible* if two transactions that access the same data item can obtain these locks on that data item at the same time. As Figure C.10 shows, read locks are compatible, whereas read-write or write-write locks are not. Therefore, it is possible, for example, for two transactions to read the same data item concurrently.

The distributed DBMS not only manages locks but also handles the lock management responsibilities on behalf of the transactions. In other words, users do not need to specify when a data item needs to be locked; the distributed DBMS takes care of that every time the transaction issues a read or write operation.

	$rl_i(x)$	$wl_i(x)$
$rl_j(x)$	compatible	not compatible
$wl_j(x)$	not compatible	not compatible

Fig. C.10: Compatibility Matrix of Lock Modes

In locking-based systems, the scheduler (see Figure C.5) is a *lock manager* (LM). The transaction manager passes to the lock manager the database operation (read or write) and associated information (such as the item that is accessed and the identifier of the transaction that issues the database operation). The lock manager then checks if the lock unit that contains the data item is already locked. If so, and if the existing lock mode is incompatible with that of the current transaction, the current operation is delayed. Otherwise, the lock is set in the desired mode and the database operation is passed on to the data processor for actual database access. The transaction manager is then informed of the results of the operation. The termination of a transaction results in the release of its locks and the initiation of another transaction that might be waiting for access to the same data item.

The locking algorithm as described above will not, unfortunately, properly synchronize transaction executions. This is because to generate serializable histories, the locking and releasing operations of transactions also need to be coordinated. We demonstrate this by an example.

Example C.18. Consider the following two transactions:

T_1: Read(x)	T_2: Read(x)
$x \leftarrow x + 1$	$x \leftarrow x * 2$
Write(x)	Write(x)
Read(y)	Read(y)
$y \leftarrow y - 1$	$y \leftarrow y * 2$
Write(y)	Write(y)
Commit	Commit

The following is a valid history that a lock manager employing the locking algorithm may generate:

$$H = \{wl_1(x), R_1(x), W_1(x), lr_1(x), wl_2(x), R_2(x), w_2(x), lr_2(x), wl_2(y), R_2(y), W_2(y), lr_2(y), wl_1(y), R_1(y), W_1(y), lr_1(y)\}$$

where $lr_i(z)$ indicates the release of the lock on z that transaction T_i holds.

Note that H is not a serializable history. For example, if prior to the execution of these transactions, the values of x and y are 50 and 20, respectively, one would expect their values following execution to be, respectively, either 102 and 38 if T_1 executes before T_2, or 101 and 39 if T_2 executes before T_1. However, the result of executing H would give x and y the values 102 and 39. Obviously, H is not serializable. ♦

The problem with history H in Example C.18 is the following. The locking algorithm releases the locks that are held by a transaction (say, T_i) as soon as the associated database command (read or write) is executed, and that lock unit (say x) no longer needs to be accessed. However, the transaction itself is locking other items (say, y), after it releases its lock on x. Even though this may seem to be advantageous from the viewpoint of increased concurrency, it permits transactions to interfere with one another, resulting in the loss of isolation and atomicity. Hence the argument for *two-phase locking* (2PL).

The two-phase locking rule simply states that no transaction should request a lock after it releases one of its locks. Alternatively, a transaction should not release a lock until it is certain that it will not request another lock. 2PL algorithms execute transactions in two phases. Each transaction has a *growing phase*, where it obtains locks and accesses data items, and a *shrinking phase*, during which it releases locks (Figure C.11). The *lock point* is the moment when the transaction has achieved all its locks but has not yet started to release any of them. Thus the lock point determines the end of the growing phase and the beginning of the shrinking phase of a transaction. It has been proven that any history generated by a concurrency control algorithm that obeys the 2PL rule is serializable [Eswaran et al., 1976].

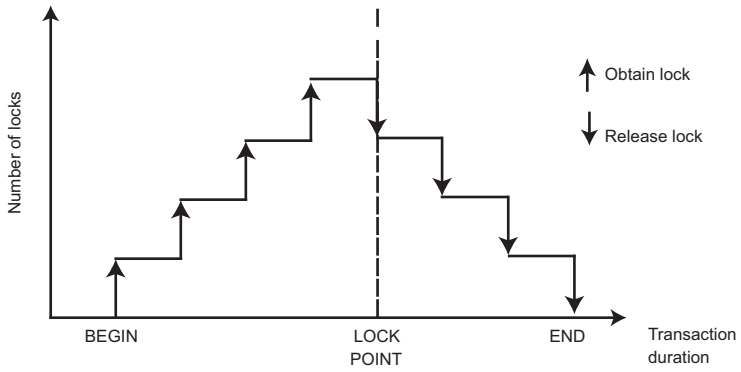

Fig. C.11: 2PL Lock Graph

Figure C.11 indicates that the lock manager releases locks as soon as access to that data item has been completed. This permits other transactions awaiting access to go ahead and lock it, thereby increasing the degree of concurrency. However, this is difficult to implement since the lock manager has to know that the transaction has obtained all its locks and will not need to lock another data item. The lock manager also needs to know that the transaction no longer needs to access the data item in question, so that the lock can be released. Finally, if the transaction aborts after it releases a lock, it may cause other transactions that may have accessed the unlocked data item to abort as well. This is known as *cascading aborts*. These problems may be overcome by *strict two-phase locking*, which releases all the locks together when

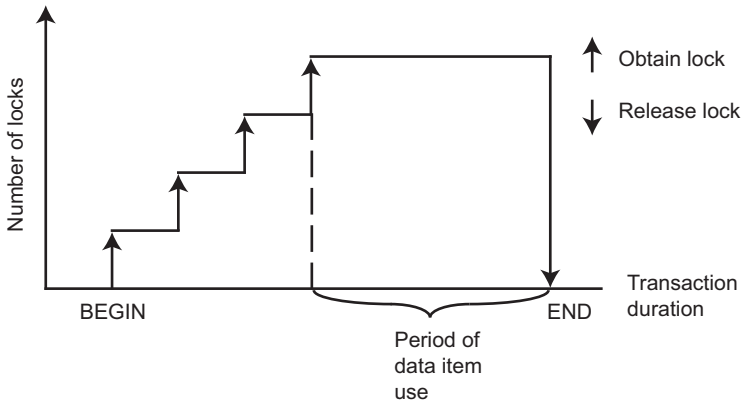

Fig. C.12: Strict 2PL Lock Graph

the transaction terminates (commits or aborts). Thus the lock graph is as shown in Figure C.12.

We should note that even though a 2PL algorithm enforces conflict serializability, it does not allow all histories that are conflict serializable. Consider the following history discussed by Agrawal and El Abbadi [1990]:

$$H = \{W_1(x), R_2(x), W_3(y), W_1(y)\}$$

H is not allowed by 2PL algorithm since T_1 would need to obtain a write lock on y after it releases its write lock on x. However, this history is serializable in the order $T_3 \rightarrow T_1 \rightarrow T_2$. The order of locking can be exploited to design locking algorithms that allow histories such as these [Agrawal and El Abbadi, 1990].

The main idea is to observe that in serializability theory, the order of serialization of conflicting operations is as important as detecting the conflict in the first place and this can be exploited in defining locking modes. Consequently, in addition to read (shared) and write (exclusive) locks, a third lock mode is defined: *ordered shared*. Ordered shared locking of an object x by transactions T_i and T_j has the following meaning: Given a history H that allows ordered shared locks between operations $o \in T_i$ and $p \in T_j$, if T_i acquires o-lock before T_j acquires p-lock, then o is executed before p. Consider the compatibility table between read and write locks given in Figure C.10. If the ordered shared mode is added, there are eight variants of this table. Figure C.10 depicts one of them and two more are shown in Figure C.13. In Figure C.13(b), for example, there is an ordered shared relationship between $rl_j(x)$ and $wl_i(x)$ indicating that T_i can acquire a write lock on x while T_j holds a read lock on x as long as the ordered shared relationship from $rl_j(x)$ to $wl_i(x)$ is observed. The eight compatibility tables can be compared with respect to their permissiveness (i.e., with respect to the histories that can be produced using them) to generate a lattice of tables such that the one in Figure C.10 is the most restrictive and the one in Figure C.13(b) is the most liberal.

	$rl_i(x)$	$wl_i(x)$
$rl_j(x)$	compatible	not compatible
$wl_j(x)$	ordered shared	not compatible

(a)

	$rl_i(x)$	$wl_i(x)$
$rl_j(x)$	compatible	ordered shared
$wl_j(x)$	ordered shared	ordered shared

(b)

Fig. C.13: Commutativity Table with Ordered Shared Lock Mode

The locking protocol that enforces a compatibility matrix involving ordered shared lock modes is identical to 2PL, except that a transaction may not release any locks as long as any of its locks are on hold. Otherwise circular serialization orders can exist.

Locking-based algorithms may cause deadlocks since they allow exclusive access to resources. It is possible that two transactions that access the same data items may lock them in reverse order, causing each to wait for the other to release its locks causing a deadlock. We discuss deadlock management in Section C.5.6.

C.5.4 Timestamp-Based Concurrency Control

Unlike locking-based algorithms, timestamp-based concurrency control algorithms do not attempt to maintain serializability by mutual exclusion. Instead, they select, a priori, a serialization order and execute transactions accordingly. To establish this ordering, the transaction manager assigns each transaction T_i a unique *timestamp*, $ts(T_i)$, at its initiation.

A timestamp is a simple identifier that serves to identify each transaction uniquely and is used for ordering. *Uniqueness* is only one of the properties of timestamp generation. The second property is *monotonicity*. Two timestamps generated by the same transaction manager should be monotonically increasing. Thus timestamps are values derived from a totally ordered domain. It is this second property that differentiates a timestamp from a transaction identifier.

With this information, it is simple to order the execution of the transactions' operations according to their timestamps. Formally, the timestamp ordering (TO) rule can be specified as follows:

TO Rule. Given two conflicting operations O_{ij} and O_{kl} belonging, respectively, to transactions T_i and T_k, O_{ij} is executed before O_{kl} if and only if $ts(T_i) < ts(T_k)$. In this case T_i is said to be the *older* transaction and T_k is said to be the *younger* one.

A scheduler that enforces the TO rule checks each new operation against conflicting operations that have already been scheduled. If the new operation belongs to a transaction that is younger than all the conflicting ones that have already been

scheduled, the operation is accepted; otherwise, it is rejected, causing the entire transaction to restart with a *new* timestamp.

A timestamp ordering scheduler that operates in this fashion is guaranteed to generate serializable histories. However, this comparison between the transaction timestamps can be performed only if the scheduler has received all the operations to be scheduled. If operations come to the scheduler one at a time (which is the realistic case), it is necessary to be able to detect, in an efficient manner, if an operation has arrived out of sequence. To facilitate this check, each data item x is assigned two timestamps: a *read timestamp* $[rts(x)]$, which is the largest of the timestamps of the transactions that have read x, and a *write timestamp* $[wts(x)]$, which is the largest of the timestamps of the transactions that have written (updated) x. It is now sufficient to compare the timestamp of an operation with the read and write timestamps of the data item that it wants to access to determine if any transaction with a larger timestamp has already accessed the same data item.

Architecturally (see Figure C.5), the transaction manager is responsible for assigning a timestamp to each new transaction and attaching this timestamp to each database operation that it passes on to the scheduler. The latter component is responsible for keeping track of read and write timestamps as well as performing the serializability check.

C.5.5 Optimistic Concurrency Control

For completeness of discussion, we mention optimistic concurrency control in this appendix, but we do not discuss this class further since they are covered in detail in the main body of the book (Chapter 5). Briefly, they differ from the locking-based and timestamp-based concurrency control in that the overriding assumption is that transactions do not conflict too much. So, they can be executed until the commit point without much synchronization and a validation is performed at that point to see if the isolation property has been violated. If that is the case, the transaction is aborted and restarted; otherwise, it is allowed to commit.

C.5.6 Deadlock Management

As we indicated before, any locking-based concurrency control algorithm may result in deadlocks, since there is mutual exclusion of access to shared resources (data) and transactions may wait on locks. Furthermore, we have seen that some TO-based algorithms that require the waiting of transactions (e.g., strict TO) may also cause deadlocks. Therefore, the distributed DBMS requires special procedures to handle them.

A deadlock can occur because transactions wait for one another. Informally, a deadlock situation is a set of requests that can never be granted by the concurrency control mechanism.

Example C.19. Consider two transactions T_i and T_j that hold write locks on two entities x and y [i.e., $wl_i(x)$ and $wl_j(y)$]. Suppose that T_i now issues a $rl_i(y)$ or a $wl_i(y)$. Since y is currently locked by transaction T_j, T_i will have to wait until T_j releases its write lock on y. However, if during this waiting period, T_j now requests a lock (read or write) on x, there will be a deadlock. This is because, T_i will be blocked waiting for T_j to release its lock on y while T_j will be waiting for T_i to release its lock on x. In this case, the two transactions T_i and T_j will wait indefinitely for each other to release their respective locks. ♦

A deadlock is a permanent phenomenon. If one exists in a system, it will not go away unless outside intervention takes place. This outside interference may come from the user, the system operator, or the software system (the operating system or the distributed DBMS).

A useful tool in analyzing deadlocks is a *wait-for graph* (WFG). A WFG is a directed graph that represents the wait-for relationship among transactions. The nodes of this graph represent the concurrent transactions in the system. An edge $T_i \rightarrow T_j$ exists in the WFG if transaction T_i is waiting for T_j to release a lock on some entity. Figure C.14 depicts the WFG for Example C.19.

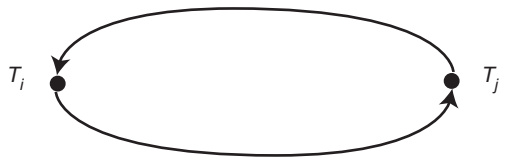

Fig. C.14: A WFG Example

Using the WFG, it is easier to indicate the condition for the occurrence of a deadlock. A deadlock occurs when the WFG contains a cycle.

There are three known methods for handling deadlocks: prevention, avoidance, and detection and resolution. In the remainder of this section we discuss each approach in more detail.

C.5.6.1 Deadlock Prevention

Deadlock prevention methods guarantee that deadlocks cannot occur in the first place. Thus the transaction manager checks a transaction when it is first initiated and does not permit it to proceed if it may cause a deadlock. To perform this check, it is required that all of the data items that will be accessed by a transaction be predeclared. The transaction manager then permits a transaction to proceed if all the data items that it

will access are available. Otherwise, the transaction is not permitted to proceed. The transaction manager reserves all the data items that are predeclared by a transaction that it allows to proceed.

Unfortunately, such systems are not very suitable for database environments. The fundamental problem is that it is usually difficult to know precisely which data items will be accessed by a transaction. Access to certain data items may depend on conditions that may not be resolved until run time. For example, in the reservation transaction that we developed in Example C.3, access to `CNAME` is conditional upon the availability of free seats. To be safe, the system would thus need to consider the maximum set of data items, even if they end up not being accessed. This would certainly reduce concurrency. Furthermore, there is additional overhead in evaluating whether a transaction can proceed safely. On the other hand, such systems require no run-time support, which reduces the overhead. It has the additional advantage that it is not necessary to abort and restart a transaction due to deadlocks. This not only reduces the overhead but also makes such methods suitable for systems that have no provisions for undoing processes.[7]

C.5.6.2 Deadlock Avoidance

Deadlock avoidance schemes either employ concurrency control techniques that will never result in deadlocks or require that potential deadlock situations are detected in advance and steps are taken such that they will not occur. We consider both of these cases.

The simplest means of avoiding deadlocks is to order the resources and insist that each process request access to these resources in that order. This solution was long ago proposed for operating systems and can be adapted to DBMS by ordering the lock units in the database so that transactions always request locks in that order. This approach is not practical, however, because, as opposed to the number of resources managed by an operating system, the number of lock units in a database are far more numerous and the database is dynamic, making the maintenance of an order impractical.

An alternative is to make use of transaction timestamps to prioritize transactions and resolve deadlocks by aborting transactions with higher (or lower) priorities. To implement this type of prevention method, the lock manager is modified as follows. If a lock request of a transaction T_i is denied, the lock manager does not automatically force T_i to wait. Instead, it applies a prevention test to the requesting transaction and the transaction that currently holds the lock (say T_j). If the test is passed, T_i is permitted to wait for T_j; otherwise, one transaction or the other is aborted.

Examples of this approach is the WAIT-DIE and WOUND-WAIT algorithms [Rosenkrantz et al., 1978], also used in the MADMAN DBMS [GE, 1976]. These algorithms are based on the assignment of timestamps to transactions. WAIT-DIE is

[7] This is not a significant advantage since most systems have to be able to undo transactions for reliability purposes, as we will see in Section C.6.

a non-preemptive algorithm in that if the lock request of T_i is denied because the lock is held by T_j, it never preempts T_j, following the rule:

> **WAIT-DIE Rule.** If T_i requests a lock on a data item that is already locked by T_j, T_i is permitted to wait if and only if T_i is older than T_j. If T_i is younger than T_j, then T_i is aborted and restarted with the same timestamp.

A preemptive version of the same idea is the WOUND-WAIT algorithm, which follows the rule:

> **WOUND-WAIT Rule.** If T_i requests a lock on a data item that is already locked by T_j, then T_i is permitted to wait if only if it is younger than T_j; otherwise, T_j is aborted and the lock is granted to T_i.

The rules are specified from the viewpoint of T_i: T_i waits, T_i dies, and T_i wounds T_j. In fact, the result of wounding and dying are the same: the affected transaction is aborted and restarted. With this perspective, the two rules can be specified as follows:

if $ts(T_i) < ts(T_j)$ **then** T_i waits **else** T_i dies (WAIT-DIE)
if $ts(T_i) < ts(T_j)$ **then** T_j is wounded **else** T_i waits (WOUND-WAIT)

Notice that in both algorithms the younger transaction is aborted. The difference between the two algorithms is whether or not they preempt active transactions. Also note that the WAIT-DIE algorithm prefers younger transactions and kills older ones. Thus an older transaction tends to wait longer and longer as it gets older. By contrast, the WOUND-WAIT rule prefers the older transaction since it never waits for a younger one. One of these methods, or a combination, may be selected in implementing a deadlock prevention algorithm.

Deadlock avoidance methods are more suitable than prevention schemes for database environments. Their fundamental drawback is that they require run-time support for deadlock management, which adds to the run-time overhead of transaction execution.

C.5.6.3 Deadlock Detection and Resolution

Deadlock detection and resolution is the most popular and best-studied method. Detection is done by studying the WFG for the formation of cycles. Resolution of deadlocks is typically done by the selection of one or more *victim* transaction(s) that will be preempted and aborted in order to break the cycles in the WFG. Under the assumption that the cost of preempting each member of a set of deadlocked transactions is known, the problem of selecting the minimum total-cost set for breaking the deadlock cycle has been shown to be a difficult (NP-complete) problem [Leung and Lai, 1979]. However, there are some factors that affect this choice [Bernstein et al., 1987]:

1. The amount of effort that has already been invested in the transaction. This effort will be lost if the transaction is aborted.
2. The cost of aborting the transaction. This cost generally depends on the number of updates that the transaction has already performed.
3. The amount of effort it will take to finish executing the transaction. The scheduler wants to avoid aborting a transaction that is almost finished. To do this, it must be able to predict the future behavior of active transactions (e.g., based on the transaction's type).
4. The number of cycles that contain the transaction. Since aborting a transaction breaks all cycles that contain it, it is best to abort transactions that are part of more than one cycle (if such transactions exist).

C.6 DBMS Reliability

We now discuss DBMS reliability. From earlier discussion in this appendix, the reader will recall that the reliability of a DBMS refers to the atomicity and durability properties of transactions.

C.6.1 Failure Modes

Designing a reliable system that can recover from failures requires identifying the types of failures with which the system has to deal. In a DBMS, we deal with three types of failures: transaction failures (aborts), system failures, and media failures.

C.6.1.1 Transaction Failures

Transactions can fail for a number of reasons. Failure can be due to an error in the transaction caused by incorrect input data as well as the detection of a present or potential deadlock. Furthermore, some concurrency control algorithms do not permit a transaction to proceed or even to wait if the data that they attempt to access are currently being accessed by another transaction. This might also be considered a failure. The usual approach to take in cases of transaction failure is to *abort* the transaction, thus resetting the database to its state prior to the start of this transaction.[8]

The frequency of transaction failures is not easy to measure. An early study reported that in System R, 3% of the transactions aborted abnormally [Gray et al., 1981]. In general, within a single application, the ratio of transactions that abort

[8] Recall that all transaction aborts are not due to failures; in some cases, application logic requires transaction aborts as in Example B.3.

themselves is rather constant, being a function of the incorrect data, the available semantic data control features, and so on. The number of transaction aborts by the DBMS due to concurrency control considerations (mainly deadlocks) is dependent on the level of concurrency (i.e., number of concurrent transactions), the interference of the concurrent applications, the granularity of locks, and so on [Härder and Reuter, 1983].

C.6.1.2 System Failures

The reasons for system failure can be traced back to a hardware or to a software failure. The ratio of hardware failures vary from study to study and range from 18% to over 50%. Studies indicate that most hardware failures are intermittent [Roth et al., 1967; Ball and Hardie, 1967]. Most of the software failures are also reported to be transient Gray [1987], suggesting that a dump and restart may be sufficient to recover without any need to "repair" the software.

Software failures are typically caused by "bugs" in the code. The estimates for the number of bugs in software vary considerably. Figures such as 0.25 bug per 1000 instructions to 10 bugs per 1000 instructions have been reported.

The important point from the perspective of this discussion is that a system failure is always assumed to result in the loss of main memory contents. Therefore, any part of the database that was in main memory buffers is lost as a result of a system failure. However, the database that is stored in secondary storage is assumed to be safe and correct. In distributed database terminology, system failures are typically referred to as *site failures*, since they result in the failed site being unreachable from other sites in the distributed system.

We typically differentiate between partial and total failures in a distributed system. *Total failure* refers to the simultaneous failure of all sites in the distributed system; *partial failure* indicates the failure of only some sites while the others remain operational. As indicated in Chapter 1, it is this aspect of distributed systems that makes them more available.

C.6.1.3 Media Failures

Media failure refers to the failures of the secondary storage devices that store the database. Such failures may be due to operating system errors, as well as to hardware faults such as head crashes or controller failures. The important point from the perspective of DBMS reliability is that all or part of the database that is on the secondary storage is considered to be destroyed and inaccessible. Duplexing of disk storage and maintaining archival copies of the database are common techniques that deal with this sort of catastrophic problem.

Media failures are frequently treated as problems local to one site and therefore not specifically addressed in the reliability mechanisms of distributed DBMSs. We consider techniques for dealing with them in Section C.6.3.1 under local recovery

management. We then turn our attention to site failures when we consider distributed recovery functions.

C.6.2 Reliability Protocols

In this section we discuss the functions performed by the recovery manager (RM) that exists at each site. These functions maintain the atomicity and durability properties of local transactions. They relate to the execution of the commands that are passed to the RM, which are **Begin_transaction**, **Read**, **Write**, **Commit**, and **Abort**. Later in this section we introduce a new command into the RM's repertoire that initiates recovery actions after a failure.

C.6.2.1 Roles of Recovery Manager and Buffer Manager

When the RM wants to read a page of data[9] on behalf of a transaction, it issues a **Fetch** command, indicating the page that it wants to read (see Figure C.5). The buffer manager checks to see if that page is already in the buffer (due to a previous fetch command from another transaction) and if so, makes it available for that transaction; if not, it reads the page from the stable database into an empty database buffer. If the buffer is full, it selects one of the buffer pages to write back to stable storage and reads the requested stable database page into that buffer page. There are a number of different algorithms by which the buffer manager may choose the buffer page to be replaced; these are discussed in standard database textbooks.

The buffer manager also provides the interface by which the RM can actually force it to write back some of the buffer pages. This can be accomplished by means of the **Flush** command, which specifies the buffer pages that the RM wants to be written back. Different RM implementations may or may not use this forced writing. This issue is discussed further in subsequent sections.

As its interface suggests, the buffer manager acts as a conduit for all access to the database via the buffers that it manages. It provides this function by fulfilling three tasks:

1. *Searching* the buffer pool for a given page;
2. If it is not found in the buffer, *allocating* a free buffer page and *loading* the buffer page with a data page that is brought in from secondary storage;
3. If no free buffer pages are available, choosing a buffer page for *replacement*.

Searching is quite straightforward. Typically, the buffer pages are shared among the transactions that execute against the database, so search is global.

---

[9] RM's unit of access may be in blocks that have sizes different from a page. However, for simplicity, we assume that the unit of access is the same.

Allocation of buffer pages is typically done dynamically. This means that the allocation of buffer pages to processes is performed as processes execute. The buffer manager tries to calculate the number of buffer pages needed to run the process efficiently and attempts to allocate that number of pages. The best known dynamic allocation method is the *working-set algorithm* [Denning, 1968, 1980].

A second aspect of allocation is fetching data pages. The most common technique is *demand paging*, where data pages are brought into the buffer as they are referenced. However, a number of operating systems prefetch a group of data pages that are in close physical proximity to the data page referenced. Buffer managers choose this route if they detect sequential access to a file.

In replacing buffer pages, the best known technique is the least recently used (LRU) algorithm that attempts to determine the *logical reference strings* [Effelsberg and Härder, 1984] of processes to buffer pages and to replace the page that has not been referenced for an extended period. The anticipation here is that if a buffer page has not been referenced for a long time, it probably will not be referenced in the near future.

The techniques discussed above are the most common. Other alternatives are discussed in [Effelsberg and Härder, 1984].

Clearly, these functions are similar to those performed by operating system (OS) buffer managers. However, quite frequently, DBMSs bypass OS buffer managers and manage disks and main memory buffers themselves due to a number of problems (see, e.g., [Stonebraker, 1981]) that are beyond the scope of this book. Basically, the requirements of DBMSs are usually incompatible with the services that OSs provide. The consequence is that DBMS kernels duplicate OS services with an implementation that is more suitable for their needs.

The RM and the buffer manager are also responsible for maintaining the DBMS log that records all actions on the database. We discuss these in the next section.

C.6.2.2 Recovery Information

In this section we assume that only system failures occur. We defer the discussion of techniques for recovering from media failures until later. Since we are dealing with centralized database recovery, communication failures are not applicable.

When a system failure occurs, the volatile database is lost. Therefore, the DBMS has to maintain some information about its state at the time of the failure in order to be able to bring the database to the state that it was in when the failure occurred. We call this information the *recovery information*.

The recovery information that the system maintains is dependent on the method of executing updates. Two possibilities are in-place updating and out-of-place updating. *In-place updating* physically changes the value of the data item in the stable database. As a result, the previous values are lost. *Out-of-place updating*, on the other hand, does not change the value of the data item in the stable database but maintains the new value separately. Of course, periodically, these updated values have to be integrated into the stable database. We should note that the reliability issues are somewhat

simpler if in-place updating is not used. However, most DBMSs use it due to its improved performance.

In-Place Update Recovery Information

Since in-place updates cause previous values of the affected data items to be lost, it is necessary to keep enough information about the database state changes to facilitate the recovery of the database to a consistent state following a failure. This information is typically maintained in a *database log*. Thus each update transaction not only changes the database but the change is also recorded in the database log (Figure C.15). The log contains information necessary to recover the database state following a failure.

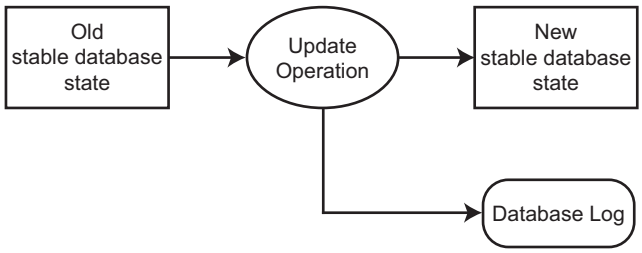

Fig. C.15: Update Operation Execution

For the following discussion assume that the RM and buffer manager algorithms are such that the buffer pages are written back to the stable database only when the buffer manager needs new buffer space. In other words, the **flush** command is not used by the RM and the decision to write back the pages into the stable database is taken at the discretion of the buffer manager. Now consider that a transaction T_1 had completed (i.e., committed) before the failure occurred. The durability property of transactions would require that the effect os T_1 be reflected in the database. However, it is possible that the volatile database pages that have been updated by T_1 may not have been written back to the stable database at the time of the failure. Therefore, upon recovery, it is important to be able to *redo* the operations of T_1. This requires some information to be stored in the database log about the effects of T_1. Given this information, it is possible to recover the database from its "old" state to the "new" state that reflects the effects of T_1 (Figure C.16).

Now consider another transaction, T_2, that was still running when the failure occurred. The atomicity property would dictate that the stable database not contain any effects of T_2. It is possible that the buffer manager may have had to write into the stable database some of the volatile database pages that have been updated by T_2.

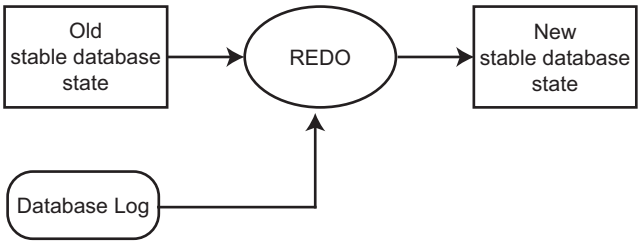

Fig. C.16: REDO Action

Upon recovery from failures it is necessary to *undo* the operations of T_2.[10] Thus the recovery information should include sufficient data to permit the undo by taking the "new" database state that reflects partial effects of T_2 and recovers the "old" state that existed at the start of T_2 (Figure C.17).

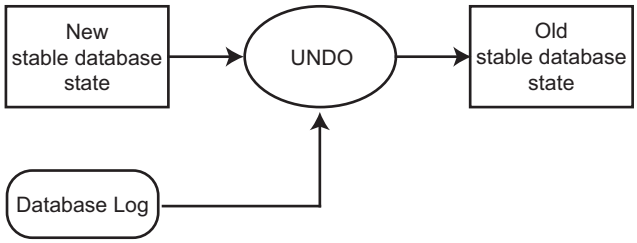

Fig. C.17: UNDO Action

We should indicate that the undo and redo actions are assumed to be idempotent. In other words, their repeated application to a transaction would be equivalent to performing them once. Furthermore, the undo/redo actions form the basis of different methods of executing the commit commands. We discuss this further in Section C.6.2.3.

The contents of the log may differ according to the implementation. However, the following minimal information for each transaction is contained in almost all database logs: a begin_transaction record, the value of the data item before the update (called the *before image*), the updated value of the data item (called the *after image*), and a termination record indicating the transaction termination condition (commit, abort). The granularity of the before and after images may be different, as it is possible to log entire pages or some smaller unit. As an alternative to this form of *state logging*,

[10] One might think that it could be possible to continue with the operation of T_2 following restart instead of undoing its operations. However, in general it may not be possible for the RM to determine the point at which the transaction needs to be restarted. Furthermore, the failure may not be a system failure but a transaction failure (i.e., T_2 may actually abort itself) after some of its actions have been reflected in the stable database. Therefore, the possibility of undoing is necessary.

operational logging, as in ARIES [Haderle et al., 1992], may be supported where the operations that cause changes to the database are logged rather than the before and after images.

The log is also maintained in main memory buffers (called *log buffers*) and written back to stable storage (called *stable log*) similar to the database buffer pages (Figure C.5). The log pages can be written to stable storage in one of two ways. They can be written *synchronously* (more commonly known as *forcing a log*) where the addition of each log record requires that the log be moved from main memory to stable storage. They can also be written *asynchronously*, where the log is moved to stable storage either at periodic intervals or when the buffer fills up. When the log is written synchronously, the execution of the transaction is suspended until the write is complete. This adds some delay to the response-time performance of the transaction. On the other hand, if a failure occurs immediately after a forced write, it is relatively easy to recover to a consistent database state.

Whether the log is written synchronously or asynchronously, one very important protocol has to be observed in maintaining logs. Consider a case where the updates to the database are written into the stable storage before the log is modified in stable storage to reflect the update. If a failure occurs before the log is written, the database will remain in updated form, but the log will not indicate the update that makes it impossible to recover the database to a consistent and up-to-date state. Therefore, the stable log is always updated prior to the updating of the stable database. This is known as the *write-ahead logging* (WAL) protocol [Gray, 1979] and can be precisely specified as follows:

1. Before a stable database is updated (perhaps due to actions of a yet uncommitted transaction), the before images should be stored in the stable log. This facilitates undo.
2. When a transaction commits, the after images have to be stored in the stable log prior to the updating of the stable database. This facilitates redo.

Out-of-Place Update Recovery Information

As we mentioned above, the most common update technique is in-place updating. Therefore, we provide only a brief overview of the other updating techniques and their recovery information. Details can be found in [Verhofstadt, 1978] and the other references given earlier.

Typical techniques for out-of-place updating are *shadowing* ([Astrahan et al., 1976; Gray, 1979]) and *differential files* [Severence and Lohman, 1976]. Shadowing uses duplicate stable storage pages in executing updates. Thus every time an update is made, the old stable storage page, called the *shadow page*, is left intact and a new page with the updated data item values is written into the stable database. The access path data structures are updated to point to the new page, which contains the current data so that subsequent accesses are to this page. The old stable storage page is retained for recovery purposes (to perform undo).

Recovery based on shadow paging is implemented in System R's recovery manager [Gray et al., 1981]. This implementation uses shadowing together with logging.

The differential file approach was discussed in Chapter 3 within the context of integrity enforcement. In general, the method maintains each stable database file as a read-only file. In addition, it maintains a corresponding read-write differential file that stores the changes to that file. Given a logical database file F, let us denote its read-only part as FR and its corresponding differential file as DF. DF consists of two parts: an insertions part, which stores the insertions to F, denoted DF^+, and a corresponding deletions part, denoted DF^-. All updates are treated as the deletion of the old value and the insertion of a new one. Thus each logical file F is considered to be a view defined as $F = (FR \cup DF^+) - DF^-$. Periodically, the differential file needs to be merged with the read-only base file.

Recovery schemes based on this method simply use private differential files for each transaction, which are then merged with the differential files of each file at commit time. Thus recovery from failures can simply be achieved by discarding the private differential files of non-committed transactions.

There are studies that indicate that the shadowing and differential files approaches may be advantageous in certain environments. One study by Agrawal and DeWitt [1985] investigates the performance of recovery mechanisms based on logging, differential files, and shadow paging, integrated with locking and optimistic (using timestamps) concurrency control algorithms. The results indicate that shadowing, together with locking, can be a feasible alternative to the more common log-based recovery integrated with locking if there are only large (in terms of the base-set size) transactions with sequential access patterns. Similarly, differential files integrated with locking can be a feasible alternative if there are medium-sized and large transactions.

C.6.2.3 Execution of RM Commands

Recall that there are five commands that form the interface to the RM. These are the **Begin_transaction, Read, Write, Commit**, and **Abort** commands. As we indicated earlier, some DBMSs do not have an explicit commit command. In this case the end (of transaction) indicator serves as the commit command. For simplicity, we specify commit explicitly.

In this section we introduce a sixth interface command to the RM: **recover**. The **Recover** command is the interface that the operating system has to the RM. It is used during recovery from system failures when the operating system asks the DBMS to recover the database to the state that existed when the failure occurred.

The execution of some of these commands (specifically, **Abort, Commit**, and **Recover**) is quite dependent on the specific RM algorithms that are used as well as on the interaction of the RM with the buffer manager. Others (i.e., **Begin_transaction, Read**, and **Write**) are quite independent of these considerations.

The fundamental design decision in the implementation of the local recovery manager, the buffer manager, and the interaction between the two components is whether or not the buffer manager obeys the local recovery manager's instructions as

to when to write the database buffer pages to stable storage. Specifically, two decisions are involved. The first one is whether the buffer manager may independently write the buffer pages updated by a transaction into stable storage during the execution of that transaction, or it waits for the RM to instruct it to write them back. We call this the *fix/no-fix* decision. The reasons for the choice of this terminology will become apparent shortly. Note that it is also called the steal/no-steal decision by Härder and Reuter [1983]. The second decision is whether the buffer manager will be forced to flush the buffer pages updated by a transaction into the stable storage at the end of that transaction (i.e., the commit point), or the buffer manager flushes them out whenever it needs to according to its buffer management algorithm. We call this the *flush/no-flush* decision. It is called the force/no-force decision by Härder and Reuter [1983].

Accordingly, four alternatives can be identified: (1) no-fix/no-flush, (2) no-fix/flush, (3) fix/no-flush, and (4) fix/flush. We will consider each of these in more detail. However, first we present the execution methods of the **Begin_transaction, Read**, and **Write** commands, which are quite independent of these considerations. Where modifications are required in these methods due to different RM and buffer manager implementation strategies, we will indicate them.

Begin_transaction, Read, and Write Commands

Begin_transaction. This command causes various components of the DBMS to carry out some bookkeeping functions. We will also assume that it causes the TM to write a begin_transaction record into the log. This is an assumption made for convenience of discussion; in reality, writing of the begin_transaction record may be delayed until the first **write** to improve performance by reducing I/O.

Read. The **Read** command specifies a data item. The RM tries to read the specified data item from the buffer pages that belong to the transaction. If the data item is not in one of these pages, it issues a **Fetch** command to the buffer manager in order to make the data available. Upon reading the data, the RM returns it to the scheduler.

Write. The **Write** command specifies the data item and the new value. As with a read command, if the data item is available in the buffers of the transaction, its value is modified in the database buffers (i.e., the volatile database). If it is not in the private buffer pages, a **Fetch** command is issued to the buffer manager, and the data is made available and updated. The before image of the data page, as well as its after image, are recorded in the log. The local recovery manager then informs the scheduler that the operation has been completed successfully.

No-fix/No-flush

This type of RM algorithm requires performing both the redo and undo operations upon recovery. It is also called redo/undo algorithm [Bernstein et al., 1987] or steal/no-force [Härder and Reuter, 1983].

Abort. As we indicated before, abort is an indication of transaction failure. Since the buffer manager may have written the updated pages into the stable database, abort will have to undo the actions of the transaction. Therefore, the RM reads the log records for that specific transaction and replaces the values of the updated data items in the volatile database with their before images. The scheduler is then informed about the successful completion of the abort action. This process is called the *transaction undo* or *partial undo*.

An alternative implementation is the use of an *abort list*, which stores the identifiers of all the transactions that have been aborted. If such a list is used, the abort action is considered to be complete as soon as the transaction's identifier is included in the abort list.

Note that even though the values of the updated data items in the stable database are not restored to their before images, the transaction is considered to be aborted at this point. The buffer manager will write the "corrected" volatile database pages into the stable database at a future time, thereby restoring it to its state prior to that transaction. An abort record is written into the log as well.

Commit. The **Commit** command causes an commit record to be written into the log by the RM. Under this scenario, no other action is taken in executing a commit command other than informing the scheduler about the successful completion of the commit action.

An alternative to writing a commit record into the log is to add the transaction's identifier to a *commit list*, which is a list of the identifiers of transactions that have committed. In this case the commit action is accepted as complete as soon as the transaction identifier is stored in this list.

Recover. The RM starts the recovery action by going to the beginning of the log and redoing the operations of each transaction for which both a begin_transaction and an commit record is found. This is called *partial redo*. Similarly, it undoes the operations of each transaction for which a begin_transaction record is found in the log without a corresponding commit record. This action is called *global undo*, as opposed to the transaction undo discussed above. The difference is that the effects of all incomplete transactions need to be rolled back, not one.

If commit list and abort list implementations are used, the recovery action consists of redoing the operations of all the transactions in the commit list and undoing the operations of all the transactions in the abort list. In the remainder of this chapter we will not make this distinction, but rather will refer to both of these recovery implementations as global undo.

No-fix/Flush

The RM algorithms that use this strategy require an explicit **Flush** command to the buffer manager for a page to be written back to stable storage. Since the RM pushes all updated pages when a transaction commits, there is no need for a redo upon recovery from a failure. These are called undo/no-redo [Bernstein et al., 1987] or steal/force [Härder and Reuter, 1983].

Abort. The execution of **Abort** is identical to the previous case. Upon transaction failure, the RM initiates a partial undo for that particular transaction.

Commit. The RM issues a **Flush** command to the buffer manager, forcing it to write back all the updated volatile database pages into the stable database. The commit command is then executed either by placing a record in the log or by insertion of the transaction identifier into the commit list as specified for the previous case. When all of this is complete, the RM informs the scheduler that the commit has been carried out successfully.

Recover. As noted above, since all the updated pages are written into the stable database at the commit point, there is no need to perform redo; all the effects of successful transactions will have been reflected in the stable database. Therefore, the recovery action initiated by the RM consists of a global undo.

Fix/No-flush

In this case the RM controls the writing of the volatile database pages into stable storage. The key here is not to permit the buffer manager to write any updated volatile database page into the stable database until at least the transaction commit point. This is accomplished by the **Fix** command, which is a modified version of the **fetch** command whereby the specified page is fixed in the database buffer and cannot be written back to the stable database by the buffer manager. Thus any **Fetch** command to the buffer manager for a write operation is replaced by a **fix** command.[11] Note that this precludes the need for a global undo operation and is therefore called a redo/no-undo algorithm [Bernstein et al., 1987] or a no-force/no-steal algorithm by [Härder and Reuter, 1983].

Abort. Since the volatile database pages have not been written to the stable database, no special action is necessary. To release the buffer pages that have been fixed by the transaction, however, it is necessary for the RM to send an **Unfix** command to the buffer manager for all such pages. It is then sufficient to carry out the abort action either by writing an abort record in the log or by including the transaction in the abort list, informing the scheduler and then forgetting about the transaction.

Commit. The RM sends an **Unfix** command to the buffer manager for every volatile database page that was previously fixed by that transaction. Note that these pages may now be written back to the stable database at the discretion of the buffer manager. The commit command is then executed either by placing an commit record in the log or by inserting the transaction identifier into the commit list as specified for the preceding case. When all of this is complete, the RM informs the scheduler that the commit has been successfully carried out.

Recover. As we mentioned above, since the volatile database pages that have been updated by ongoing transactions are not yet written into the stable database, there is no necessity for global undo. The RM, therefore, initiates a partial redo action

[11] Of course, any page that was previously fetched for read but is now being updated also needs to be fixed.

to recover those transactions that may have already committed, but whose volatile database pages may not have yet written into the stable database.

Fix/Flush

This is the case where the RM forces the buffer manager to write the updated volatile database pages into the stable database at precisely the commit point—not before and not after. This strategy is called no-undo/no-redo [Bernstein et al., 1987] or no-steal/force [Härder and Reuter, 1983].

Abort. The execution of **Abort** is identical to that of the fix/no-flush case.

Commit. The RM sends an **Unfix** command to the buffer manager for every volatile database page that was previously fixed by that transaction. It then issues a **Flush** command to the buffer manager, forcing it to write back all the unfixed volatile database pages into the stable database.[12] Finally, the **Commit** command is processed by either writing an commit record into the log or by including the transaction in the commit list. The important point to note here is that all three of these operations have to be executed as an atomic action. One step that can be taken to achieve this atomicity is to issue only a **Fush** command, which serves to unfix the pages as well. This eliminates the need to send two messages from the RM to the buffer manager, but does not eliminate the requirement for the atomic execution of the flush operation and the writing of the database log. The RM then informs the scheduler that the **Commit** has been carried out successfully. Methods for ensuring this atomicity are beyond the scope of our discussion (see [Bernstein et al., 1987]).

Recover. The **recover** command does not need to do anything in this case. This is true since the stable database reflects the effects of all the successful transactions and none of the effects of the uncommitted transactions.

C.6.3 Checkpointing

In most of the RM implementation strategies, the execution of the recovery action requires searching the entire log. This is a significant overhead because the RM is trying to find all the transactions that need to be undone and redone. The overhead can be reduced if it is possible to build a wall which signifies that the database at that point is up-to-date and consistent. In that case, the redo has to start from that point on and the undo only has to go back to that point. This process of building the wall is called *checkpointing*.

Checkpointing is achieved in three steps [Gray, 1979]:

---

[12] Our discussion here gives the impression that two commands (*Unfix* and *Flush*) need to be sent to the BM by the RM for each commit action. We have chosen to explain the action in this way only for presentation simplicity. In reality, it is, of course, possible and preferable to implement one command that instructs the BM to both unfix and flush, thereby reducing the message overhead between DBMS components.

1. Write a begin_checkpoint record into the log.
2. Collect the checkpoint data into the stable storage.
3. Write an end_checkpoint record into the log.

The first and the third steps enforce the atomicity of the checkpointing operation. If a system failure occurs during checkpointing, the recovery process will not find an end_checkpoint record and will consider checkpointing not completed.

There are a number of different alternatives for the data that is collected in Step 2, how it is collected, and where it is stored. We will consider one example here, called *transaction-consistent checkpointing* [Gray, 1979; Gray et al., 1981]. The checkpointing starts by writing the begin_checkpoint record in the log and stopping the acceptance of any new transactions by the RM. Once the active transactions are all completed, all the updated volatile database pages are flushed to the stable database followed by the insertion of an end_checkpoint record into the log. In this case, the redo action only needs to start from the end_checkpoint record in the log. The undo action can go the reverse direction, starting from the end of the log and stopping at the end_checkpoint record.

Transaction-consistent checkpointing is not the most efficient algorithm, since a significant delay is experienced by all the transactions. There are alternative checkpointing schemes such as action-consistent checkpoints, fuzzy checkpoints, and others [Gray, 1979; Lindsay, 1979].

C.6.3.1 Handling Media Failures

As we mentioned before, the previous discussion on centralized recovery considered non-media failures, where the database as well as the log stored in the stable storage survive the failure. Media failures may either be quite catastrophic, causing the loss of the stable database or of the stable log, or they can simply result in partial loss of the database or the log (e.g., loss of a track or two).

The methods that have been devised for dealing with this situation are again based on duplexing. To cope with catastrophic media failures, an *archive* copy of both the database and the log is maintained on a different (tertiary) storage medium. Thus the DBMS deals with three levels of memory hierarchy: the main memory, random access secondary storage, and tertiary storage (Figure C.5). To deal with less catastrophic failures, having duplicate copies of the database and log may be sufficient.

When a media failure occurs, the database is recovered from the archive copy by redoing and undoing the transactions as stored in the archive log. The real question is how the archive database is stored. If we consider the large sizes of current databases, the overhead of writing the entire database to tertiary storage is significant. Two methods that have been proposed for dealing with this are to perform the archiving activity concurrent with normal processing and to archive the database incrementally as changes occur so that each archive version contains only the changes that have occurred since the previous archiving.