K.D.K COLLEGE OF ENGINEERING,NAGPUR

Database Management Systems

Unit 5

Unit 5

5.1 Transaction

The term transaction refers to a collection of operations that form a single logical unit of work. For instance, transfer of money from one account to another is a transaction consisting of two updates, one to each account

Often, a collection of several operations on the database appears to be a single unit from the point of view of the database user. For example, a transfer of funds from a checking account to a savings account is a single operation from the customer's standpoint; within the database system, however, it consists of several operations. Clearly, it is essential that all these operations occur, or that, in case of a failure, none occur. It would be unacceptable if the checking account were debited, but the savings account were not credited. Collections of operations that form a single logical unit of work are called **transactions**. A database system must ensure proper execution of transactions despite failures— either the entire transaction executes, or none of it does. Furthermore, it must manage concurrent execution of transactions in a way that avoids the introduction of inconsistency. In our funds-transfer example, a transaction computing the customer's total money might see the checking-account balance before it is debited by the funds transfer transaction, but see the savings balance after it is credited. As a result, it would obtain an incorrect result.

To ensure integrity of the data, we require that the database system maintain the following properties of the transactions:

Atomicity. Either all operations of the transaction are reflected properly in the database, or none are.

Consistency. Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database.

Isolation. Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions *Ti* and *Tj* , it appears to *Ti* that either *Tj* finished execution before *Ti* started, or *Tj* started execution after *Ti* finished. Thus, each transaction is unaware of other transactions executing concurrently in the system.

Durability. After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

These properties are often called the **ACID properties**; the acronym is derived from the first letter of each of the four properties.

5.2 Transaction State

In the absence of failures, all transactions complete successfully. However, as we noted earlier, a transaction may not always complete its execution successfully. Such a transaction is termed **aborted**. If we are to ensure the atomicity property, an aborted transaction must have no effect on the state of the database. Thus, any changes that the aborted transaction made to the database must be undone. Once the changes caused by an aborted transaction have been undone, we say that the transaction has been **rolled back**. It is part of the responsibility of the recovery scheme to manage transaction aborts. A transaction that completes its execution successfully is said to be **committed**. A committed transaction that has performed updates transforms the database into a new consistent state, which must persist even if there is a system failure. Once a transaction has committed, we cannot undo its effects by aborting it. The only way to undo the effects of a committed transaction is to execute a **compensating transaction**. For instance, if a transaction added $20 to an account, the compensating transaction would subtract $20 from the account. However, it is not always possible to create such a compensating transaction. Therefore, the responsibility of writing and executing a compensating transaction is left to the user, and is not handled by the database system.

We therefore establish a simple abstract transaction model. A transaction must be in one of the following states:

- **Active**, the initial state; the transaction stays in this state while it is executing
- **Partially committed**, after the final statement has been executed
- **Failed**, after the discovery that normal execution can no longer proceed
- **Aborted**, after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction
- **Committed**, after successful completion

The state diagram corresponding to a transaction appears in Figure. We say that a transaction has committed only if it has entered the committed state. Similarly, we say that a transaction has aborted only if it has entered the aborted state. A transaction is said to have **terminated** if has either committed or aborted. A transaction starts in the active state. When it finishes its final statement, it enters the partially committed state. At this point, the transaction has completed its execution, but it is still possible that it may have to be aborted, since the actual output may still be temporarily residing in main memory, and thus a hardware failure may preclude its successful completion. The database system then writes out enough information to disk that, even in the event of a failure, the updates performed by the transaction can be re-created when the system restarts after the failure. When the last of this information is written out, the transaction enters the committed state.

A transaction enters the failed state after the system determines that the transaction can no longer proceed with its normal execution (for example, because of hardware or logical errors). Such a transaction must be rolled back. Then, it enters the aborted state. At this point, the system has two options:

It can **restart** the transaction, but only if the transaction was aborted as a result of some hardware or software error that was not created through the internal logic of the transaction. A restarted transaction is considered to be a new transaction.

It can **kill** the transaction. It usually does so because of some internal logical error that can be corrected only by rewriting the application program, or because the input was bad, or because the desired data were not found in the database.

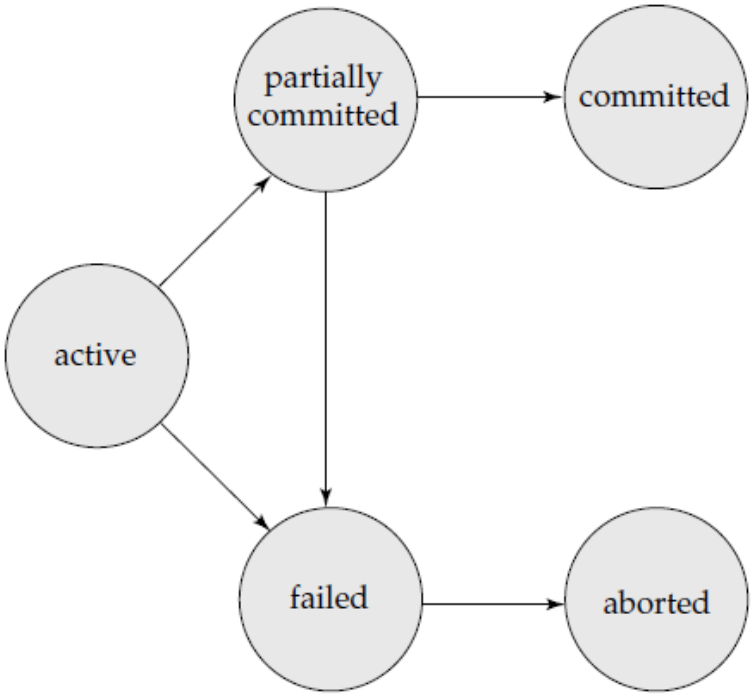


Figure: State diagram of transaction

5.3 Implementation of Atomicity and Durability

The recovery-management component of a database system can support atomicity and durability by a variety of schemes. We first consider a simple, but extremely inefficient, scheme called the **shadow copy** scheme. This scheme, which is based on making copies of the database, called *shadow* copies, assumes that only one transaction is active at a time. The scheme also assumes that the database is simply a file on disk. A pointer called db-pointer is maintained on disk; it points to the current copy of the database.

In the shadow-copy scheme, a transaction that wants to update the database first creates a complete copy of the database. All updates are done on the new database copy, leaving the original copy, the **shadow copy**, untouched. If at any point the transaction has to be aborted, the system merely deletes the new copy. The old copy of the database has not been affected.

If the transaction completes, it is committed as follows. First, the operating system is asked to make sure that all pages of the new copy of the database have been written out to disk. (Unix systems use the flush command for this purpose.) After the operating system has written all the pages to disk, the database system updates the pointer db-pointer to point to the new copy of the database; the new copy then becomes the current copy of the database. The old copy of the database is then deleted. Figure depicts the scheme, showing the database state before and after the update

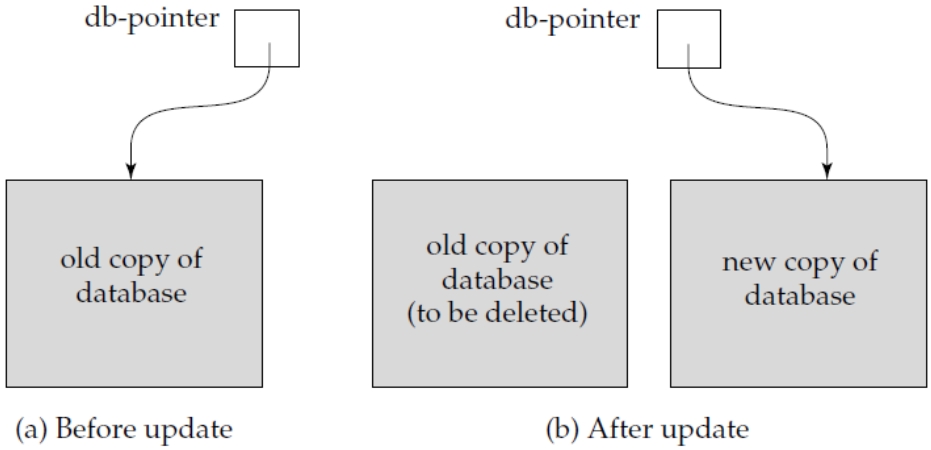


Figure : Shadow-copy technique for atomicity and durability.

5.4 Concurrent Executions

Transaction-processing systems usually allow multiple transactions to run concurrently. Allowing multiple transactions to update data concurrently causes several complications with consistency of the data, as we saw earlier. Ensuring consistency in spite of concurrent execution of transactions requires extra work; it is far easier to insist that transactions run serially—that is, one at a time, each starting only after the previous one has completed. However, there are two good reasons for allowing concurrency:

• **Improved throughput and resource utilization**. A transaction consists of many steps. Some involve I/O activity; others involve CPU activity. The CPU and the disks in a computer system can operate in parallel. Therefore, I/O activity can be done in parallel with processing at the CPU. The parallelism of the CPU and the I/O system can therefore be exploited to run multiple transactions in parallel. While a read or write on behalf of one transaction is in progress on one disk, another transaction can be running in the CPU, while another disk may be executing a read or write on behalf of a third transaction. All of this increases the **throughput** of the system—that is, the number of transactions executed in a given amount of time. Correspondingly, the processor and disk **utilization** also increase; in other words, the processor and disk spend less time idle, or not performing any useful work.

• **Reduced waiting time**. There may be a mix of transactions running on a system, some short and some long. If transactions run serially, a short transaction may have to wait for a preceding

long transaction to complete, which can lead to unpredictable delays in running a transaction. If the transactions are operating on different parts of the database, it is better to let them run concurrently, sharing the CPU cycles and disk accesses among them. Concurrent execution reduces the unpredictable delays in running transactions. Moreover, it also reduces the **average response time**: the average time for a transaction to be completed after it has been submitted.
The database system must control the interaction among the concurrent transactions to prevent them from destroying the consistency of the database. It does so through a variety of mechanisms called **concurrency-control schemes**.

Let $T1$ and $T2$ be two transactions that transfer funds from one account to another. Transaction $T1$ transfers \$50 from account A to account B. It is defined as

$T1$: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

Transaction $T2$ transfers 10 percent of the balance from account A to account B. It is defined as

$T2$: read(A);
$temp := A * 0.1$;
$A := A - temp$;
write(A);
read(B);
$B := B + temp$;
write(B).

Suppose the current values of accounts A and B are \$1000 and \$2000, respectively. Suppose also that the two transactions are executed one at a time in the order $T1$ followed by $T2$. This execution sequence appears in Figure. In the figure, the sequence of instruction steps is in chronological order from top to bottom, with instructions of $T1$ appearing in the left column and instructions of $T2$ appearing in the right column. The final values of accounts A and B, after the execution in Figure takes place, are \$855 and \$2145, respectively. Thus, the total amount of money in accounts A and B—that is, the sum $A + B$—is preserved after the execution of both transactions. Similarly, if the transactions are executed one at a time in the order $T2$ followed by $T1$, then the corresponding execution sequence is that of Figure. Again, as expected, the sum $A + B$ is preserved, and the final values of accounts A and B are \$850 and \$2150, respectively. The execution sequences just described are called **schedules**. They represent the chronological order in which instructions are executed in the system

$T1$	$T2$
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Figure: Schedule 1—a serial schedule in which $T1$ is followed by $T2$.

The execution sequences just described are called **schedules**. They represent the chronological order in which instructions are executed in the system. Clearly, a schedule for a set of transactions must consist of all instructions of those transactions, and must preserve the order in which the instructions appear in each individual transaction. These schedules are **serial**: Each serial schedule consists of a sequence of instructions from various transactions, where the instructions belonging to one single transaction appear together in that schedule. Thus, for a set of *n* transactions, there exist *n*! different valid serial schedules

T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	

Figure : Schedule 2—a serial schedule in which T_2 is followed by T_1.

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

Figure: Schedule 3—a concurrent schedule equivalent to schedule 1.

Suppose that the two transactions are executed concurrently. One possible schedule appears in above Figure. After this execution takes place, we arrive at the same state as the one in which the transactions are executed serially in the order $T1$ followed by $T2$. The sum$A + B$ is indeed preserved.

5.5 Serializability

Types Of Serializability:
1)conflict Serializability
2)view Serializability

5.5.1 Conflict Serializability

Let us consider a schedule S in which there are two consecutive instructions Ii and Ij, of transactions Ti and Tj , respectively ($i \neq j$). If Ii and Ij refer to different data items, then we can swap Ii and Ij without affecting the results of any instruction in the schedule. However, if Ii and Ij refer to the same data item Q, then the order of the two steps maymatter. Since we are dealing with only read and write instructions, there are four cases that we need to consider:

1. Ii = read(Q), Ij = read(Q). The order of Ii and Ij does not matter, since the same value of Q is read by Ti and Tj , regardless of the order.

2. Ii = read(Q), Ij = write(Q). If Ii comes before Ij, then Ti does not read the value of Q that is written by Tj in instruction Ij. If Ij comes before Ii, then Ti reads the value of Q that is written by Tj. Thus, the order of Ii and Ij matters.

3. Ii = write(Q), Ij = read(Q). The order of Ii and Ij matters for reasons similar to those of the previous case.

4. Ii = write(Q), Ij = write(Q). Since both instructions are write operations, the order of these instructions does not affect either Ti or Tj . However, the value obtained by the next read(Q) instruction of S is affected, since the result of only the latter of the two write instructions is preserved in the database. If there is no other write(Q) instruction after Ii and Ij in S, then the order of Ii and Ij directly affects the final value of Q in the database state that results from schedule S.

Thus, only in the case where both Ii and Ij are read instructions does the relative order of their execution not matter. We say that Ii and Ij **conflict** if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation.

To illustrate the concept of conflicting instructions, we consider schedule 3, in Figure

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Figure

Schedule 3—showing only the read and write instructions.

The write(*A*) instruction of *T*1 conflicts with the read(*A*) instruction of *T*2. However, the write(*A*) instruction of *T*2 does not conflict with the read(*B*) instruction of *T*1, because the two instructions access different data items. Let *Ii* and *Ij* be consecutive instructions of a schedule *S*. If *Ii* and *Ij* are instructions of different transactions and *Ii* and *Ij* do not conflict, then we can swap the order of *Ii* and *Ij* to produce a new schedule *S*_. We expect *S* to be equivalent to *S*_, since all instructions appear in the same order in both schedules except for *Ii* and *Ij*, whose order does not matter. Since the write(*A*) instruction of *T*2 in schedule 3 of Figure 15.7 does not conflict with the read(*B*) instruction of *T*1, we can swap these instructions to generate an equivalent schedule, schedule 5, in Figure 15.8. Regardless of the initial system state, schedules 3 and 5 both produce the same final system state.

T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

Figure Schedule 5—schedule 3 after swapping of a pair of instructions.

We continue to swap nonconflicting instructions:

- Swap the read(*B*) instruction of *T*1 with the read(*A*) instruction of *T*2.
- Swap the write(*B*) instruction of *T*1 with the write(*A*) instruction of *T*2.
- Swap the write(*B*) instruction of *T*1 with the read(*A*) instruction of *T*2.

The final result of these swaps, schedule 6 of Figure 15.9, is a serial schedule. Thus, we have shown that schedule 3 is equivalent to a serial schedule.

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Figure Schedule 6—a serial schedule that is equivalent to schedule 3.

If a schedule *S* can be transformed into a schedule *S*_ by a series of swaps of nonconflicting instructions, we say that *S* and *S*_ are **conflict equivalent**. In our previous examples, schedule 1 is not conflict equivalent to schedule 2.However, schedule 1 is conflict equivalent to schedule 3, because the read(*B*) and write(*B*) instruction of *T*1 can be swapped with the read(*A*) and write(*A*)

instruction of *T*2. The concept of conflict equivalence leads to the concept of conflict serializability. We say that a schedule *S* is **conflict serializable** if it is conflict equivalent to a serial schedule. Thus, schedule 3 is conflict serializable, since it is conflict equivalent to the serial schedule 1.

5.5.2 View Serializability

Consider two schedules *S* and *S*_, where the same set of transactions participates in both schedules. The schedules *S* and *S*_ are said to be **view equivalent** if three conditions are met:

1. For each data item *Q*, if transaction *Ti* reads the initial value of *Q* in schedule *S*, then transaction *Ti* must, in schedule *S*_, also read the initial value of *Q*.

2. For each data item *Q*, if transaction *Ti* executes read(*Q*) in schedule *S*, and if that value was produced by a write(*Q*) operation executed by transaction *Tj* , then the read(*Q*) operation of transaction *Ti* must, in schedule *S*_, also read the value of *Q* that was produced by the same write(*Q*) operation of transaction *Tj* .

3. For each data item *Q*, the transaction (if any) that performs the final write(*Q*) operation in schedule *S* must perform the final write(*Q*) operation in schedule *S*_. Conditions 1 and 2 ensure that each transaction reads the same values in both schedules and, therefore, performs the same computation. Condition 3, coupled with conditions 1 and 2, ensures that both schedules result in the same final system state. In our previous examples, schedule 1 is not view equivalent to schedule 2, since, in schedule 1, the value of account *A* read by transaction *T*2 was produced by *T*1, whereas this case does not hold in schedule 2. However, schedule 1 is view equivalent to schedule 3, because the values of account *A* and *B* read by transaction *T*2 were produced by *T*1 in both schedules. The concept of view equivalence leads to the concept of view serializability We say that a schedule *S* is **view serializable** if it is view equivalent to a serial schedule. As an illustration, suppose that we augment schedule 7 with transaction *T*6, and obtain schedule 9 in Figure. Schedule 9 is view serializable. Indeed, it is view equivalent to the serial schedule <*T*3, *T*4, *T*6>, since the one read(*Q*) instruction reads the initial value of *Q* in both schedules, and *T*6 performs the final write of *Q* in both schedules.

Every conflict-serializable schedule is also view serializable, but there are viewserializable schedules that are not conflict serializable. Indeed, schedule 9 is not conflict serializable, since every pair of consecutive instructions conflicts, and, thus, no swapping of instructions is possible.

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

Figure Schedule 9—a view-serializable schedule.

Observe that, in schedule 9, transactions *T*4 and *T*6 perform write(*Q*) operations without having performed a read(*Q*) operation. Writes of this sort are called **blind writes**. Blind writes appear in any view-serializable schedule that is not conflict serializable.

5.6 Recoverability

We address the effect of transaction failures during concurrent execution. If a transaction *Ti* fails, for whatever reason, we need to undo the effect of this transaction to ensure the atomicity property of the transaction. In a system that allows concurrent execution, it is necessary also to ensure that any transaction *Tj* that is dependent on *Ti* (that is, *Tj* has read data written by *Ti*) is also aborted. To achieve this surety, we need to place restrictions on the type of schedules permitted in the system.

5.6.1 Recoverable Schedules

Consider schedule 11 in Figure

T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

Figure Schedule 11.

in which *T*9 is a transaction that performs only one instruction: read(*A*). Suppose that the system allows *T*9 to commit immediately after executing the read(*A*) instruction. Thus, *T*9 commits before *T*8 does. Now suppose that *T*8 fails before it commits. Since *T*9 has read the value of data item *A* written by *T*8, we must abort *T*9 to ensure transaction atomicity. However, *T*9 has already committed and cannot be aborted. Thus, we have a situation where it is impossible to recover correctly from the failure of *T*8. Schedule 11, with the commit happening immediately after the read(*A*) instruction, is an example of a *nonrecoverable* schedule, which should not be allowed. Most database system require that all schedules be *recoverable*. A **recoverable schedule** is one where, for each pair of transactions *Ti* and *Tj* such that *Tj* reads a data item previously written by *Ti*, the commit operation of *Ti* appears before the commit operation of *Tj* .

5.6.2 Cascadeless Schedules

Even if a schedule is recoverable, to recover correctly from the failure of a transaction *Ti*, we may have to roll back several transactions. Such situations occur if transactions have read data written by *Ti*. As an illustration, consider the partial schedule of Figure

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

Figure Schedule 12.

Transaction *T*10 writes a value of *A* that is read by transaction *T*11. Transaction *T*11 writes a value of *A* that is read by transaction *T*12. Suppose that, at this point, *T*10 fails. *T*10 must be rolled back. Since *T*11 is dependent on *T*10, *T*11 must be rolled back. Since *T*12 is dependent on *T*11, *T*12 must be rolled back. This phenomenon, in which a single transaction failure leads to a series of transaction rollbacks, is called **cascading rollback**. Cascading rollback is undesirable, since it leads to the undoing of a significant amount of work. It is desirable to restrict the schedules to those where cascading rollbacks cannot occur. Such schedules are called *cascadeless* schedules. Formally, a **cascadeless schedule** is one where, for each pair of transactions *Ti* and *Tj* such that *Tj* reads a data item previously written by *Ti*, the commit operation of *Ti* appears before the read operation of *Tj* . It is easy to verify that every cascadeless schedule is also recoverable.

5.7 Concurrency control

Why Concurrency Control Is Needed

Several problems can occur when concurrent transactions execute in an uncontrolled manner.We illustrate some of these problems by referring to a much simplified airline reservations database in which a record is stored for each airline flight. Each record includes the number of reserved seats on that flight as a *named data item,* among other information.

(a)

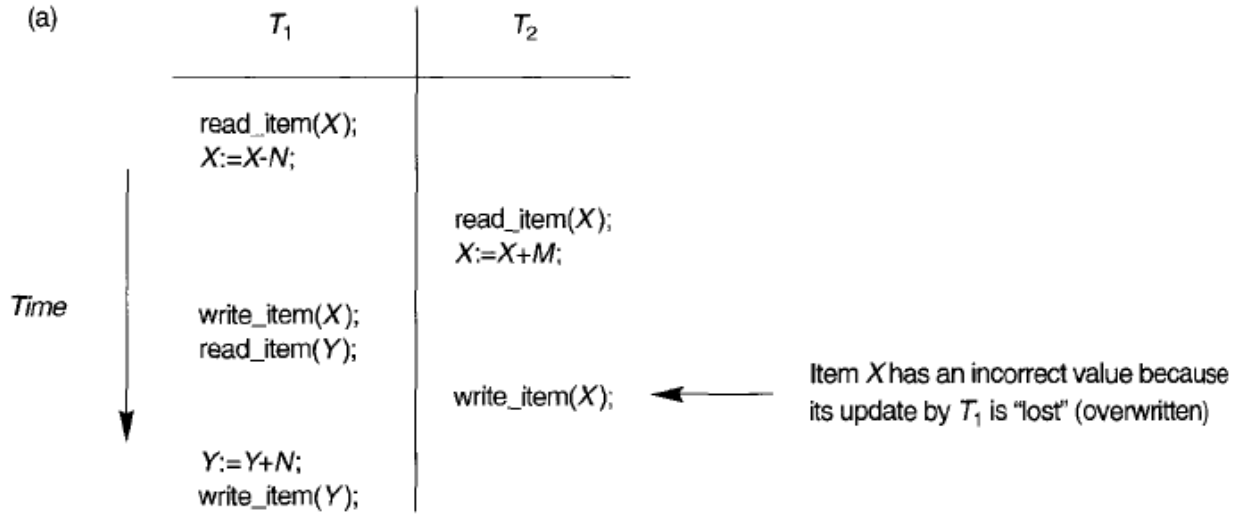
T_1
read_item (X);
X:=X-N;
write_item (X);
read_item (Y);
Y:=Y+N;
write_item (Y);

(b)

T_2
read_item (X);
X:=X+M;
write_item (X);

FIGURE Two sample transactions. (a) Transaction *Tl* . (b) Transaction *Tz.*

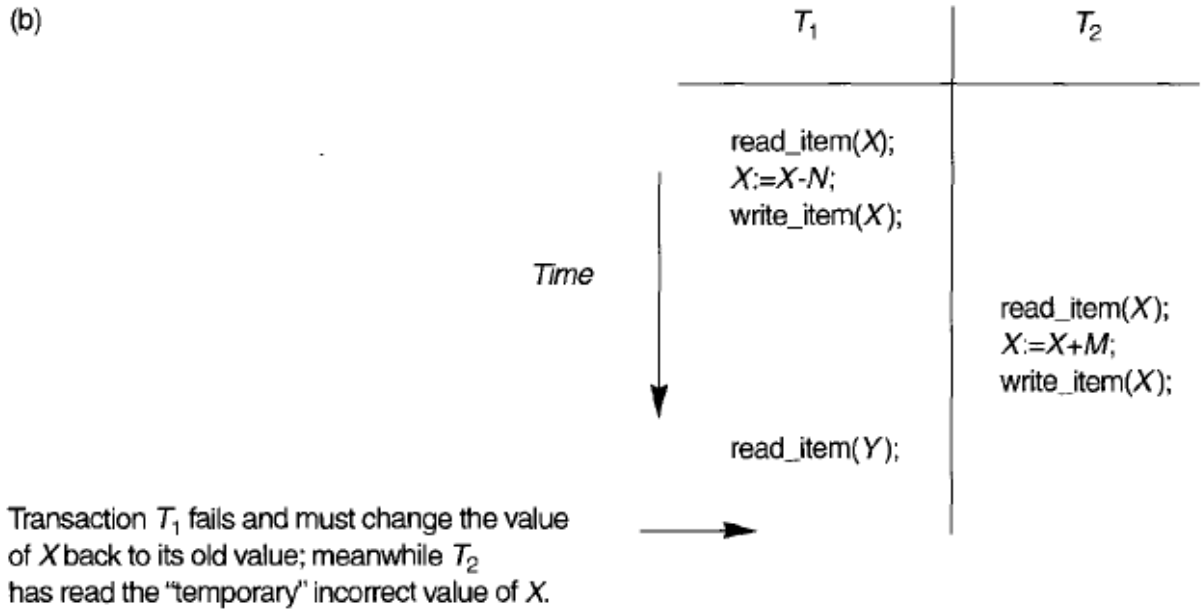
Figure a shows a transaction T j that *transfers* N reservations from one flight whose number of reserved seats is stored in the database item named X to another flight whose number of reserved seats is stored in the database item named Y. Figure b shows a simpler transaction Tz that just *reserves* M seats on the first flight (X) referenced in transaction Tj When a database access program is written, it has the flight numbers, their dates, and the number of seats to be booked as parameters; hence, the same program can be used to execute many transactions, each with different flights and numbers of seats to be booked. For concurrency control purposes, a transaction is a *particular execution* of a program on a specific date, flight, and number of seats. In Figure a and b, the transactions T j and Tz are *specific executions* of the programs that refer to the specific flights whose numbers of seats are stored in data items X and Y in the database. We now discuss the types of problems we may encounter with these two transactions if they run concurrently. The Lost Update Problem. This problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database items incorrect. Suppose that transactions T j and Tz are submitted at approximately the same time, and suppose that their operations are interleaved as shown in Figure a



(a) The lost update problem.

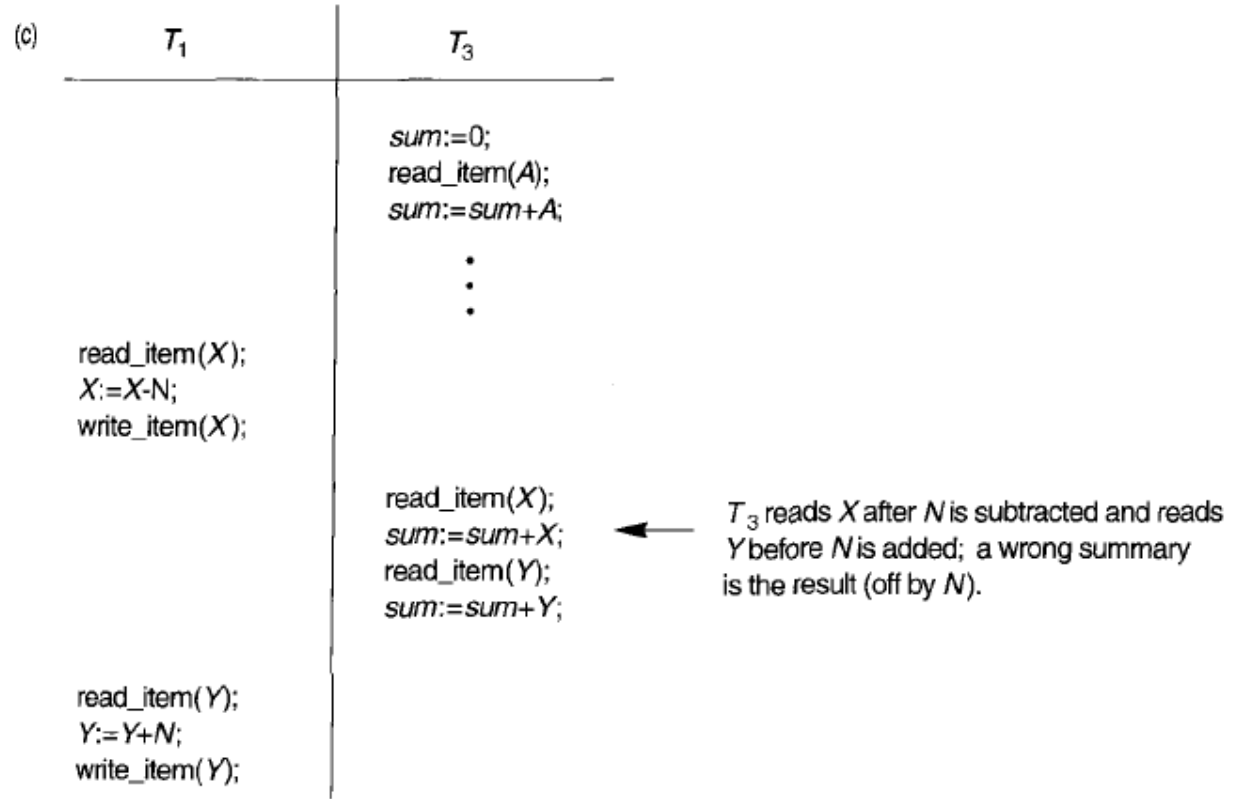
then the final value of item X is incorrect, because Tz reads the value ofX *before* T j changes it in the database, and hence the updated value resulting from T j is lost. For example, ifX = 80 at the start (originally there were 80 reservations on the flight), N = *5 (T)* transfers 5 seat reservations from the flight corresponding to X to the flight corresponding to *Y),* and M = 4 *(Tz* reserves 4 seats on X), the final result should be X = 79; but in the interleaving of operations shown in Figure 17.3a, it is X = 84 because the update in T j that removed the five seats from X was *lost.* The Temporary Update (or Dirty Read) Problem. This problem occurs when one transaction updates a database item and then the transaction fails for some reason. The updated

item is accessed by another transaction before it is changed back to its original value.



(b) The temporary update problem.

Figure b shows an example where T1 updates item X and thenfails before completion, so the system must change X back to its original value. Before it can do so, however, transaction T2 reads the "temporary" value of X, which will not be recorded permanently in the database because of the failure of T r-The value of item X that is read by T2 is called *dirty data*, because it has been created by a transaction that has not completedand committed yet; hence, this problem is also known as the *dirty read problem.* The Incorrect Summary Problem. If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated. For example, suppose that a transaction T} is calculating the total number of reservations on all the flights; meanwhile, transaction T1 is executing.



c)The incorrect summary problem.

If the interleaving of operations shown in Figure 17.3c occurs, the result of *T3* will be off by an amount N because T} reads the value of X *after* N seats have been subtractedfrom it but reads the value of *Y before* those N seats have been added to it. Another problem that may occur is called unrepeatable read, where a transaction T reads an item twice and the item is changed by another transaction T' between the two reads. Hence, T receives *different values* for its two reads of the same item.

Deadlock Handling

A system is in a deadlock state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set. More precisely, there exists a set of waiting transactions *{T*0, *T*1, . . ., *Tn}* such that *T*0 is waiting for a data item that *T*1 holds, and *T*1 is waiting for a data item that *T*2 holds, and . . ., and *Tn*−1 is waiting for a data item that *Tn* holds, and *Tn* is waiting for a data item that *T*0 holds. None of the transactions can make progress in such a situation. There are two principal methods for dealing with the deadlock problem. We can use a **deadlock prevention** protocol to ensure that the system will *never* enter a deadlock state. Alternatively, we can allow the system to enter a deadlock state, and then try to recover by using a **deadlock detection** and **deadlock recovery** scheme.

Deadlock Prevention

There are two approaches to deadlock prevention. One approach ensures that no cyclic waits can occur by ordering the requests for locks, or requiring all locks to be acquired together. The other approach is closer to deadlock recovery, and performs transaction rollback instead of waiting for a lock, whenever thewait could potentially result in a deadlock. The simplest scheme under the first approach requires that each transaction locks all its data items before it begins execution. Moreover, either all are locked in one step or none are locked. There are two main disadvantages to this protocol:

(1) it is often hard to predict, before the transaction begins, what data items need to be locked;

(2) data-item utilization may be very low, since many of the data items may be locked but unused for a long time. Another approach for preventing deadlocks is to impose an ordering of all data items, and to require that a transaction lock data items only in a sequence consistent with the ordering. the approach is to use a total order of data items, in conjunction with two-phase locking. Once a transaction has locked a particular item, it cannot request locks on items that precede that item in the ordering. This scheme is easy to implement, as long as the set of data items accessed by a transaction is known when the transaction starts execution. The second approach for preventing deadlocks is to use preemption and transaction rollbacks. In preemption, when a transaction *T*2 requests a lock that transaction *T*1 holds, the lock granted to *T*1 may be **preempted** by rolling back of *T*1, and granting of the lock to *T*2. To control the preemption, we assign a unique timestamp to each transaction. The system uses these timestamps only to decide whether a transaction should wait or roll back. Locking is still used for concurrency control. If a transaction is rolled back, it retains its *old* timestamp when restarted. Two different deadlockprevention schemes using timestamps have been proposed:

1. The **wait–die** scheme is a nonpreemptive technique. When transaction *Ti* requests a data item currently held by *Tj* , *Ti* is allowed to wait only if it has a timestamp smaller than that of *Tj* (that is, *Ti* is older than *Tj*). Otherwise, *Ti* is rolled back (dies).

For example, suppose that transactions *T*22, *T*23, and *T*24 have timestamps 5, 10, and 15, respectively. If *T*22 requests a data itemheld by *T*23, then *T*22 will wait. If *T*24 requests a data item held by *T*23, then *T*24 will be rolled back.

2. The **wound–wait** scheme is a preemptive technique. It is a counterpart to the wait–die scheme. When transaction *Ti* requests a data item currently held by *Tj* , *Ti* is allowed to wait only if it has a timestamp larger than that of *Tj* (that is, *Ti* is younger than *Tj*). Otherwise, *Tj* is rolled back (*Tj* is *wounded* by *Ti*). Returning to our example, with transactions *T*22, *T*23, and *T*24, if *T*22 requests a data item held by *T*23, then the data item will be preempted from *T*23, and *T*23 will be rolled back. If *T*24 requests a data item held by *T*23, then *T*24 will wait. Whenever the system rolls back transactions, it is important to ensure that there is no **starvation**—that is, no transaction gets rolled back repeatedly and is never allowed to make progress. Both the wound–wait and the wait–die schemes avoid starvation: At any time, there is a transaction with the smallest timestamp. significant differences in the way that the two schemes operate.

• In the wait–die scheme, an older transaction must wait for a younger one to release its data item. Thus, the older the transaction gets, the more it tends to wait. By contrast, in the wound–wait scheme, an older transaction never waits for a younger transaction. In the wait–die scheme, if a transaction *Ti* dies and is rolled back because it requested a data item held by transaction *Tj*, then *Ti* may reissue the same sequence of requests when it is restarted. If the data item is still held by *Tj* , then *Ti* will die again. Thus, *Ti* may die several times before acquiring the needed data item. Contrast this series of events with what happens in the wound–wait scheme. Transaction *Ti* is wounded and rolled back because *Tj* requested a data item that it holds. When *Ti* is restarted and requests the data item now being held by *Tj* , *Ti* waits. Thus, there may be fewer rollbacks in the wound–wait scheme.

The major problem with both of these schemes is that unnecessary rollbacks may occur.

Deadlock Detection and Recovery

If a system does not employ some protocol that ensures deadlock freedom, then a detection and recovery scheme must be used. An algorithm that examines the state of the system is invoked periodically to determine whether a deadlock has occurred. If one has, then the system must attempt to recover from the deadlock. To do so, the system must:

• Maintain information about the current allocation of data items to transactions, as well as any outstanding data item requests.
• Provide an algorithm that uses this information to determine whether the system has entered a deadlock state.
• Recover from the deadlock when the detection algorithm determines that a deadlock exists.

Deadlock Detection

Deadlocks can be described precisely in terms of a directed graph called a **wait-for graph**. This graph consists of a pair $G = (V, E)$, where V is a set of vertices and E is a set of edges. The set of vertices consists of all the transactions in the system. Each element in the set E of edges is an ordered pair $Ti \rightarrow Tj$. If $Ti \rightarrow Tj$ is in E, then there is a directed edge from transaction *Ti* to *Tj* , implying that transaction *Ti* is waiting for transaction *Tj* to release a data item that it needs.
When transaction *Ti* requests a data item currently being held by transaction *Tj* , then the edge $Ti \rightarrow Tj$ is inserted in the wait-for graph. This edge is removed only when transaction *Tj* is no longer holding a data item needed by transaction *Ti*. A deadlock exists in the system if and only if the wait-for graph contains a cycle. Each transaction involved in the cycle is said to be deadlocked. To detect deadlocks, the system needs to maintain the wait-for graph, and periodically to invoke an algorithm that searches for a cycle in the graph.

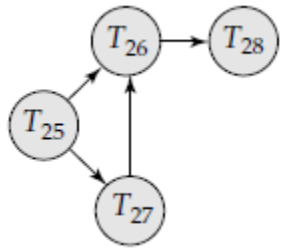


Figure Wait-for graph with no cycle.

To illustrate these concepts, consider the wait-for graph in Figure, which depicts the following situation:
• Transaction *T*25 is waiting for transactions *T*26 and *T*27.
• Transaction *T*27 is waiting for transaction *T*26.
• Transaction *T*26 is waiting for transaction *T*28.

Since the graph has no cycle, the system is not in a deadlock state. Suppose now that transaction *T*28 is requesting an item held by *T*27. The edge *T*28 → *T*27 is added to the wait-for graph, resulting in the new system state in Figure 16.19. This time, the graph contains the cycle

*T*26 → *T*28 →*T*27 →*T*26

implying that transactions *T*26, *T*27, and *T*28 are all deadlocked.

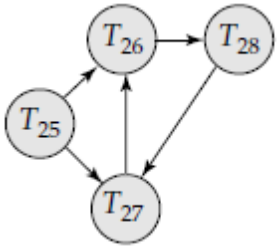


Figure Wait-for graph with a cycle.

Recovery from Deadlock
When a detection algorithm determines that a deadlock exists, the system must **recover** from the deadlock. The most common solution is to roll back one or more transactions to break the deadlock. Three actions need to be taken:

1. Selection of a victim. Given a set of deadlocked transactions, we must determine which transaction (or transactions) to roll back to break the deadlock.We should roll back those transactions that will incur the minimum cost. Unfortunately, the term *minimum cost* is not a precise one. Many factors may determine the cost of a rollback, including

a. How long the transaction has computed, and how much longer the transaction will compute before it completes its designated task.
b. How many data items the transaction has used.

c. How many more data items the transaction needs for it to complete.
d. How many transactions will be involved in the rollback.

2. Rollback. Once we have decided that a particular transaction must be rolled back, we must determine how far this transaction should be rolled back. The simplest solution is a **total rollback**: Abort the transaction and then restart it. However, it is more effective to roll back the transaction only as far as necessary to break the deadlock. Such **partial rollback** requires the system to maintain additional information about the state of all the running transactions.

3. Starvation. In a system where the selection of victims is based primarily on cost factors, it may happen that the same transaction is always picked as a victim. As a result, this transaction never completes its designated task, thus there is **starvation**. We must ensure that transaction can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

Recovery system

The various data items in the database may be stored and accessed in a number of different storage media. To understand how to ensure the atomicity and durability properties of a transaction, we must gain a better understanding of these storage media and their access methods.

Storage Types
storage media can be distinguished by their relative speed, capacity, and resilience to failure, and classified as volatile storage or nonvolatile storage.

Volatile storage. Information residing in volatile storage does not usually survive system crashes. Examples of such storage are main memory and cache memory. Access to volatile storage is extremely fast, both because of the speed of the memory access itself, and because it is possible to access any data item in volatile storage directly.

• **Nonvolatile storage**. Information residing in nonvolatile storage survives system crashes. Examples of such storage are disk and magnetic tapes. Disks are used for online storage, whereas tapes are used for archival storage. Both, however, are subject to failure (for example, head crash), which may result in loss of information. At the current state of technology, nonvolatile storage is slower than volatile storage by several orders of magnitude.

Stable storage. Information residing in stable storage is *never* lost

Stable-Storage Implementation

To implement stable storage, we need to replicate the needed information in several nonvolatile storage media (usually disk) with independent failure modes, and to update the information in a controlled manner to ensure that failure during data transfer does not damage the needed information. Block transfer between memory and disk storage can result in

• **Successful completion**. The transferred information arrived safely at its destination.
• **Partial failure**. A failure occurred in the midst of transfer, and the destination block has incorrect information.
• **Total failure**. The failure occurred sufficiently early during the transfer that the destination block remains intact.

We require that, if a **data-transfer failure** occurs, the system detects it and invokes a recovery procedure to restore the block to a consistent state. To do so, the system must maintain two physical blocks for each logical database block; in the case of mirrored disks, both blocks are at the same location; in the case of remote backup, one of the blocks is local, whereas the other is at a remote site. An output operation is executed as follows:

1. Write the information onto the first physical block.
2. When the first write completes successfully, write the same information onto the second physical block.

3. The output is completed only after the second write completes successfully. During recovery, the system examines each pair of physical blocks. If both are the same and no detectable error exists, then no further actions are necessary. If the system detects an error in one block, then it replaces its content with the content of the other block. If both blocks contain no detectable error, but they differ in content, then the system replaces the content of the first block with the value of the second. This recovery procedure ensures that a write to stable storage either succeeds completely (that is, updates all copies) or results in no change.

Data Access
Blocks are the units of data transfer to and from disk, and may contain several data items.We shall assume that no data item spans two or more blocks. Transactions input information from the disk to main memory, and then output the information back onto the disk. The input and output operations are done in block units. The blocks residing on the disk are referred to as **physical blocks**; the blocks residing temporarily in main memory are referred to as **buffer blocks**. The area of memory where blocks reside temporarily is called the **disk buffer**. Block movements between disk and main memory are initiated through the following two operations:

1. input(*B*) transfers the physical block *B* to main memory.
2. output(*B*) transfers the buffer block *B* to the disk, and replaces the appropriate physical block there.

Figure illustrates this scheme.

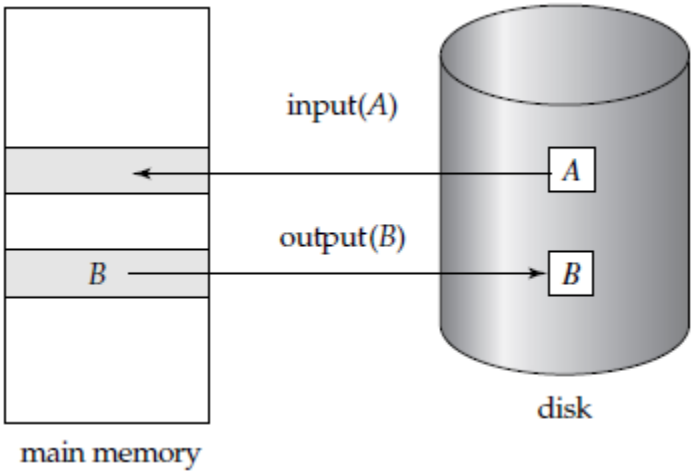


Figure Block storage operations.

Each transaction *Ti* has a private work area in which copies of all the data items accessed and updated by *Ti* are kept. The system creates this work area when the transaction is initiated; the system removes it when the transaction either commits or aborts. Each data item *X* kept in the work area of transaction *Ti* is denoted by *xi*. Transaction *Ti* interacts with the database system by transferring data to and from its work area to the system buffer.We transfer data by these two operations:

1. read(*X*) assigns the value of data item *X* to the local variable *xi*. It executes this operation as follows:

a. If block *BX* on which *X* resides is not in main memory, it issues input(*BX*).

b. It assigns to *xi* the value of *X* from the buffer block.

2. write(*X*) assigns the value of local variable *xi* to data item *X* in the buffer block.

It executes this operation as follows:

a. If block *BX* on which*X* resides is not in main memory, it issues input(*BX*).

b. It assigns the value of *xi* to *X* in buffer *BX*.

Note that both operations may require the transfer of a block from disk to main memory. They do not, however, specifically require the transfer of a block from main memory to disk A buffer block is eventually written out to the disk either because the buffer manager needs the memory space for other purposes or because the database system wishes to reflect the change to *B* on the disk. We shall say that the database system performs a **force-output** of buffer *B* if it issues an output(*B*). When a transaction needs to access a data item *X* for the first time, it must execute read(*X*). The system then performs all updates to *X* on *xi*. After the transaction accesses *X* for the final time, it must execute write(*X*) to reflect the change to *X* in the database itself. The output(*BX*) operation for the buffer block *BX* on which *X* resides does not need to take effect immediately after write(*X*) is executed, since the block *BX* may contain other data items that are still being accessed. Thus, the actual output may take place later. Notice that, if the system crashes after the write(*X*) operation was executed but before output(*BX*) was executed, the new value of *X* is never written to disk and, thus, is lost.