Contents

Index

# 1

# THE PROBLEM

## 1.1 TRANSACTIONS

Concurrency control is the activity of coordinating the actions of processes that operate in parallel, access shared data, and therefore potentially interfere with each other. Recovery is the activity of ensuring that software and hardware failures do not corrupt persistent data. Concurrency control and recovery problems arise in the design of hardware, operating systems, real time systems, communications systems, and database systems, among others. In this book, we will explore concurrency control and recovery problems in database systems.

We will study these problems using a model of database systems. This model is an abstraction of many types of data handling systems, such as database management systems for data processing applications, transaction processing systems for airline reservations or banking, and file systems for a general purpose computing environment. Our study of concurrency control and recovery applies to any such system that conforms to our model.

The main component of this model is the *transaction*. Informally, a transaction is an execution of a program that accesses a shared database. The goal of concurrency control and recovery is to ensure that transactions execute *atomically*, meaning that

*1.* each transaction accesses shared data without interfering with other transactions, and

*2.* if a transaction terminates normally, then all of its effects are made permanent; otherwise it has no effect at all.

The purpose of this chapter is to make this model precise.

In this section we present a user-oriented model of the system, which consists of a database that a user can access by executing transactions. In Section 1.2, we explain what it means for a transaction to execute atomically in the presence of failures. In Section 1.3, we explain what it means for a transaction to execute atomically in an environment where its database accesses can be interleaved with those of other transactions. Section 1.4 presents a model of a database system's concurrency control and recovery components, whose goal is to realize transaction atomicity.

### Database Systems

A *database* consists of a set of named *data items*. Each data item has a *value*. The values of the data items at any one time comprise the *state* of the database.

In practice, a data item could be a word of main memory, a page of a disk, a record of a file, or a field of a record. The size of the data contained in a data item is called the *granularity* of the data item. Granularity will usually be unimportant to our study and we will therefore leave it unspecified. When we leave granularity unspecified, we denote data items by lower case letters, typically $x$, $y$, and $z$.

A *database system* (*DBS*)[1] is a collection of hardware and software modules that support commands to access the database, called *database operations* (or simply *operations*). The most important operations we will consider are Read and Write. Read($x$) returns the value stored in data item $x$. Write($x$, *val*) changes the value of $x$ to *val*. We will also use other operations from time to time.

The DBS executes each operation *atomically*. This means that the DBS behaves as if it executes operations *sequentially*, that is, one at a time. To obtain this behavior, the DBS might *actually* execute operations sequentially. However, more typically it will execute operations *concurrently*. That is, there may be times when it is executing more than one operation at once. However, even if it executes operations concurrently, the final effect must be the same as some sequential execution.

For example, suppose data items $x$ and $y$ are stored on two different devices. The DBS might execute operations on $x$ and $y$ in this order:

1. execute Read($x$);
2. after step (1) is finished, concurrently execute Write($x$, 1) and Read($y$);
3. after step (2) is finished, execute Write($y$, 0).

Although Write($x$, 1) and Read($y$) were executed concurrently, they may be regarded as having executed atomically. This is because the execution just

[1] We use the abbreviation *DBS*, instead of the more conventional *DBMS*, to emphasize that a DBS in our sense may be much less than an integrated database management system. For example, it may only be a simple file system with transaction management capabilities.

given has the same effect as a sequential execution, such as Read($x$), Write($x$, 1), Read($y$), Write($y$, 0).

The DBS also supports *transaction operations*: Start, Commit, and Abort. A program tells the DBS that it is about to begin executing a new transaction by issuing the operation *Start*. It indicates the termination of the transaction by issuing either the operation *Commit* or the operation *Abort*. By issuing a Commit, the program tells the DBS that the transaction has terminated normally and all of its effects should be made permanent. By issuing an Abort, the program tells the DBS that the transaction has terminated abnormally and all of its effects should be obliterated.

A program must issue each of its database operations on behalf of a particular transaction. We can model this by assuming that the DBS responds to a Start operation by returning a unique transaction identifier. The program then attaches this identifier to each of its database operations, and to the Commit or Abort that it issues to terminate the transaction. Thus, from the DBS's viewpoint, a transaction is defined by a Start operation, followed by a (possibly concurrent) execution of a set of database operations, followed by a Commit or Abort.

A transaction may be a *concurrent* execution of two or more programs. That is, the transaction may submit two operations to the DBS before the DBS has responded to either one. However, the transaction's last operation *must* be a Commit or Abort. Thus, the DBS must refuse to process a transaction's database operation if it arrives after the DBS has already executed the transaction's Commit or Abort.

### Transaction Syntax

Users interact with a DBS by invoking programs. From the user's viewpoint, a *transaction* is the execution of one or more programs that include database and transaction operations.

For example, consider a banking database that contains a file of customer accounts, called Accounts, each entry of which contains the balance in one account. A useful transaction for this database is one that transfers money from one account to another.

```
Procedure Transfer begin
    Start;
    input(fromaccount, toaccount, amount);
    /* This procedure transfers "amount" from "fromaccount" into "toaccount." */
    temp := Read(Accounts[fromaccount]);
    if temp < amount then begin
        output("insufficient funds");
        Abort
    end
    else begin
        Write(Accounts[fromaccount], temp − amount);
        temp := Read(Accounts[toaccount]);
        Write(Accounts[toaccount], temp + amount);
        Commit;
        output("transfer completed");
    end;
    return
end
```

"Transfer" illustrates the programming language we will use in examples. It includes the usual procedure declaration (**Procedure** procedure-name **begin** procedure-body **end**), assignment statement (variable := expression), a conditional statement (**if** Boolean-expression **then** statement **else** statement), **input** (which reads a list of values from a terminal or other input device and assigns them to variables), **output** (which lists values of constants or variables on a terminal or other output device), **begin-end** brackets to treat a statement list as a single statement (**begin** statement-list **end**), a statement to return from a procedure (**return**), and brackets to treat text as a comment (/* comment */). We use semicolons as statement separators, in the style of Algol and Pascal.

The choice of language for expressing transactions is not important to our study of concurrency control and recovery. In practice, the language could be a database query language, a report writing language, or a high level programming language augmented with database operations. No matter how the transaction is expressed, it must eventually be translated into programs that issue database operations, since database operations are the only way to access the database. We therefore assume that the programs that comprise transactions are written in a high level language with embedded database operations.

Transfer is an unrealistic program in that it doesn't perform any error checking, such as testing for incorrect input. Although such error checking is essential if application programs are to be reliable, it is unimportant to our understanding of concurrency control and recovery problems. Therefore, to keep our example programs short, we will ignore error checking in those programs.

### Commit and Abort

After the DBS executes a transaction's Commit (or Abort) operation, the transaction is said to be *committed* (or *aborted*). A transaction that has issued its Start operation but is not yet committed or aborted is called *active*. A transaction is *uncommitted* if it is aborted or active.

A transaction issues an Abort operation if it cannot be completed correctly. The transaction itself may issue the Abort because it has detected an error from which it cannot recover, such as the "insufficient funds" condition in Transfer.

Or the Abort may be "imposed" on a transaction by circumstances beyond its control.

For example, suppose a system failure interrupts the execution of a Transfer transaction after it debited one account but before it credited the other. Assuming Transfer's internal state was lost as a consequence of the failure, it cannot continue its execution. Therefore, when the system recovers, the DBS should cause this execution of Transfer to abort. In such cases, we still view the Abort to be an operation of the *transaction*, even though the *DBS* actually invoked the operation.

Even in the absence of system failures, the DBS may decide unilaterally to abort a transaction. For example, the DBS may discover that it has returned an incorrect value to transaction $T$ in response to $T$'s Read. It may discover this error long after it actually processed the Read. (We'll see some examples of how this may happen in the next section.) Once it discovers the error, it's too late to change the incorrect value, so it must abort $T$.

When a transaction aborts, the DBS wipes out all of its effects. The prospect that a transaction may be aborted calls for the ability to determine a point in time after which the DBS guarantees to the user that the transaction will *not* be aborted and its effects will be permanent. For example, in processing a deposit through an automatic teller machine, a customer does not want to leave the machine before being assured that the deposit transaction will not be aborted. Similarly, from the bank's viewpoint, in processing a withdrawal the teller machine should not dispense any money before making certain that the withdrawal transaction will not be aborted.

The Commit operation accomplishes this guarantee. Its invocation signifies that a transaction terminated "normally" and that its effects should be permanent. Executing a transaction's Commit constitutes a guarantee by the DBS that it will not abort the transaction and that the transaction's effects will survive subsequent failures of the system.

Since the DBS is at liberty to abort a transaction $T$ until $T$ commits, the user can't be sure that $T$'s output will be permanent as long as $T$ is active. Thus, a user should not trust $T$'s output until the DBS tells the user that $T$ has committed. This makes Commit an important operation for read-only transactions (called *queries*) as well as for transactions that write into the database (called *update transactions* or *updaters*).

The DBS should guarantee the permanence of Commit under the weakest possible assumptions about the correct operation of hardware, systems software, and application software. That is, it should be able to handle as wide a variety of errors as possible. At least, it should ensure that data written by committed transactions is not lost as a consequence of a computer or operating system failure that corrupts main memory but leaves disk storage unaffected.

### Messages

We assume that each transaction is self-contained, meaning that it performs its computation without any direct communication with other transactions. Transactions do communicate indirectly, of course, by storing and retrieving data in the database. However, this is the *only* way they can affect each other's execution.

To ensure transaction atomicity, the DBS must control all of the ways that transactions interact. This means that the DBS must mediate each transaction's operations that can affect other transactions. In our model, the only such operations are accesses to shared data. Since a transaction accesses shared data by issuing database operations to the DBS, the DBS can control all such actions, as required.

In many systems, transactions are allowed to communicate by sending messages. We allow such message communication in our model, provided that those messages are stored in the database. A transaction sends or receives a message by writing or reading the data item that holds the message.

This restriction on message communication only applies to messages *between* transactions. Two or more processes that are executing on behalf of the same transaction can freely exchange messages, and those messages need not be stored in the database. In general, a transaction is free to control its internal execution using any available mechanism. Only interactions between different transactions need to be controlled by the DBS.

## 1.2 RECOVERABILITY

The recovery system should make the DBS behave as if the database contains all of the effects of committed transactions and none of the effects of uncommitted ones. If transactions never abort, recovery is rather easy. Since all transactions eventually commit, the DBS simply executes database operations as they arrive. So to understand recovery, one must first look at the processing of Aborts.

When a transaction aborts, the DBS must wipe out its effects. The effects of a transaction $T$ are of two kinds: effects on data, that is, values that $T$ wrote in the database; and effects on other transactions, namely, transactions that read values written by $T$. Both should be obliterated.

The DBS should remove $T$'s effects by restoring, for each data item $x$ updated by $T$, the value $x$ would have had if $T$ had never taken place. We say that the DBS *undoes* $T$'s Write operations.

The DBS should remove $T$'s effects by aborting the affected transactions. Aborting these transactions may trigger further abortions, a phenomenon called *cascading abort*.

For example, suppose the initial values of $x$ and $y$ are 1, and suppose transactions $T_1$ and $T_2$ issue operations that the DBS executes in the following order:

Write$_1$($x$, 2); Read$_2$($x$); Write$_2$($y$, 3).

The subscript on each Read and Write denotes the transaction that issued it. Now, suppose $T_1$ aborts. Then the DBS undoes Write$_1$($x$, 2), restoring $x$ to the value 1. Since $T_2$ read the value of $x$ written by $T_1$, $T_2$ must be aborted too, a cascading abort. So, the DBS undoes Write$_2$($y$, 3), restoring $y$ to 1.

Recall that by committing a transaction, the DBS guarantees that it will not subsequently abort the transaction. Given the possibility of cascading aborts, the DBS must be careful when it makes that guarantee. Even if a transaction $T$ issues its Commit, the DBS may still need to abort $T$, because $T$ may yet be involved in a cascading abort. This will happen if $T$ read a data item from some transaction that subsequently aborts. Therefore, $T$ cannot commit until all transactions that wrote values read by $T$ are guaranteed not to abort, that is, are themselves committed. Executions that satisfy this condition are called *recoverable*.

This is an important concept so let's be more precise. We say a *transaction $T_j$ reads x from transaction $T_i$ in an execution*, if

1. $T_j$ reads $x$ after $T_i$ has written into it;
2. $T_i$ does not abort before $T_j$ reads $x$; and
3. every transaction (if any) that writes $x$ between the time $T_i$ writes it and $T_j$ reads it, aborts before $T_j$ reads it.

A *transaction $T_j$ reads from $T_i$* if $T_j$ reads some data item from $T_i$. An execution is *recoverable* if, for every transaction $T$ that commits, $T$'s Commit follows the Commit of every transaction from which $T$ read.

Recoverability is required to ensure that aborting a transaction does not change the semantics of committed transactions' operations. To see this, let's slightly modify our example of cascading aborts:

Write$_1$($x$, 2); Read$_2$($x$); Write$_2$($y$, 3); Commit$_2$.

This is not a recoverable execution, because $T_2$ read $x$ from $T_1$ and yet the Commit of $T_2$ does *not* follow the Commit of $T_1$ (which is still active). The problem is what to do if $T_1$ now aborts. We can leave $T_2$ alone, which would violate the semantics of $T_2$'s Read($x$) operation; Read$_2$($x$) actually returned the value 2, but given that $T_1$ has aborted, it should have returned the value that $x$ had before Write$_1$($x$, 2) executed. Alternatively, we can abort $T_2$, which would violate the semantics of $T_2$'s Commit. Either way we are doomed. However, if the DBS had delayed Commit$_2$, thus making the execution recoverable, there would be no problem with aborting $T_2$. The system, not having processed $T_2$'s Commit, never promised that it would not abort $T_2$. In general, delaying the processing of certain Commits is one way the DBS can ensure that executions are recoverable.

### Terminal I/O

Intuitively, an execution is recoverable if the DBS is always able to reverse the effects of an aborted transaction on other transactions. The definition of recoverable relies on the assumption that all such effects are through Reads and Writes. Without this assumption, the definition of recoverable does not correspond to its intuition.

There is one other type of interaction between transactions that calls the definition into question, namely, interactions through users. A transaction can interact with a terminal or other user-to-computer I/O device using **input** and **output** statements. Since a user can read the output of one transaction and, using that information, select information to feed as input to another transaction, **input** and **output** statements are another method by which transactions can indirectly communicate.

For example, suppose a transaction $T_1$ writes output to a terminal before it commits. Suppose a user reads that information on the terminal screen, and based on it decides to enter some input to another transaction $T_2$. Now suppose $T_1$ aborts. Indirectly, $T_2$ is executing operations based on the output of $T_1$ Since $T_1$ has aborted, $T_2$ should abort too, a cascading abort. Unfortunately, the DBS doesn't know about this dependency between $T_1$ and $T_2$, and therefore isn't in a position to ensure automatically that the cascading abort takes place.

In a sense, the error here is really the user's. Until the DBS writes the message "Transaction $T_1$ has committed" on the user's terminal, the user should not trust the output produced by $T_1$. Until that message appears, the user doesn't know whether $T_1$ will commit; it may abort and thereby invalidate its terminal output. In the previous paragraph, the user incorrectly assumed $T_1$'s terminal output would be committed, and therefore prematurely propagated $T_1$'s effects to another transaction.

The DBS can prevent users from prematurely propagating the effects of an uncommitted transaction $T$ by *deferring* $T$'s output statements until after $T$ commits. Then the user will only see committed output.

It is often acceptable for the DBS to adopt this deferred output approach. In particular, it works well if each transaction requests all of its input from the user before it produces any output. But if a transaction $T$ writes a message to a terminal and subsequently requests input from the user, deferring output puts the user in an untenable position. The user's response to $T$'s input request may depend on the uncommitted output that he or she has not yet seen. In this case, the DBS must release the output to the terminal before $T$ commits.

Suppose the DBS does release $T$'s output and the user then responds to $T$'s input request. Now suppose $T$ aborts. Depending on the reason why $T$ aborted, the user may choose to try executing $T$ again. Since other transactions may have executed between the time $T$ aborted and was restarted, $T$'s second execution may be reading a different database state than its first execution. It may therefore produce different output, which may suggest to the user that different input is required than in $T$'s first execution. Therefore, in reexecuting $T$, the DBS *cannot* reuse the terminal input from $T$'s first execution.

### Avoiding Cascading Aborts

Enforcing recoverability does not remove the possibility of cascading aborts. On the contrary, cascading aborts may have to take place precisely to guarantee that an execution is recoverable. Let's turn to our example again:

Write$_1(x, 2)$; Read$_2(x)$; Write$_2(y, 3)$; Abort$_1$.

This is a recoverable execution. $T_2$ must abort because if it *ever* committed, the execution would no longer be recoverable.

However, the prospect of cascading aborts is unpleasant. First, they require significant bookkeeping to keep track of which transactions have read from which others. Second, and more importantly, they entail the possibility of uncontrollably many transactions being forced to abort because some other transaction happened to abort. This is very undesirable. In practice, DBSs are designed to avoid cascading aborts.

We say that a DBS *avoids cascading aborts* (or is *cascadeless*) if it ensures that every transaction reads only those values that were written by committed transactions. Thus, only committed transactions can affect other transactions.

To achieve cascadelessness, the DBS must delay each Read($x$) until all transactions that have previously issued a Write($x$, *val*) have either aborted or committed. In doing so, recoverability is also achieved: a transaction must execute its Commit after having executed all its Reads and therefore after all the Commits of transactions from which it read.

### Strict Executions

Unfortunately, from a practical viewpoint, avoiding cascading aborts is not always enough. A further restriction on executions is often desirable. To motivate this, consider the question of undoing a transaction's Writes. Intuitively, for each data item $x$ that the transaction wrote, we want to restore the value $x$ would have had if the transaction had never taken place. Let's make this more precise. Take any execution involving a transaction $T$ that wrote into $x$. Suppose $T$ aborts. If we assume that the execution avoids cascading aborts, no other transaction needs to be aborted. Now erase from the execution in question all operations that belong to $T$. This results in a new execution. "The value that $x$ would have had if $T$ had never occurred" is precisely the value of $x$ in this new execution.

For example, consider

Write$_1(x, 1)$; Write$_1(y, 3)$; Write$_2(y, 1)$; Commit$_1$; Read$_2(x)$; Abort$_2$.

The execution that results if we erase the operations of $T_2$ is

Write$_1$($x$, 1); Write$_1$($y$, 3); Commit$_1$.

The value of $y$ after this execution is obviously 3. This is the value that should be restored for $y$ when $T_2$ aborts in the original execution.

The *before image* of a Write($x$, *val*) operation in an execution is the value that $x$ had just before this operation. For instance, in our previous example the before image of Write$_2$($y$, 1) is 3. It so happens that this is also the value that should be restored for $y$ when $T_2$ (which issued Write$_2$($y$, 1)) aborts. It is very convenient to implement Abort by restoring the before images of all Writes of a transaction. Many DBSs work this way. Unfortunately, this is not always correct, unless some further assumptions are made about executions. The following example illustrates the problems.

Suppose the initial value of $x$ is 1. Consider the execution

Write$_1$($x$, 2); Write$_2$($x$, 3); Abort$_1$.

The before image of Write$_1$($x$, 2) is 1, the initial value of $x$. Yet the value of $x$ that should be "restored" when $T_1$ aborts is 3, the value written by $T_2$. This is a case where aborting $T_1$ should not really affect $x$, because $x$ was overwritten after it was written by $T_1$. Notice that there is no cascading abort here, because $T_2$ wrote $x$ without having previously read it.

To take the example further, suppose that $T_2$ now aborts as well. That is, we have

Write$_1$($x$, 2); Write$_2$($x$, 3); Abort$_1$; Abort$_2$.

The before image of Write$_2$($x$, 3) is 2, the value written by $T_1$. However, the value of $x$ after Write$_2$($x$, 3) is undone should be 1, the initial value of $x$ (since both updates of $x$ have been aborted). In this case the problem is that the before image was written by an aborted transaction.

This example illustrates discrepancies between the values that should be restored when a transaction aborts and the before images of the Writes issued by that transaction. Such discrepancies arise when two transactions, neither of which has terminated, have both written into the same data item. Note that if $T_1$ had aborted before $T_2$ wrote $x$ (that is, if Abort$_1$ and Write$_2$($x$, 3) were interchanged in the previous example) there would be no problem. The before image of Write$_2$($x$, 3) would then be 1, not 2, since the transaction that wrote 2 would have already aborted. Thus when $T_2$ aborts, the before image of Write$_2$($x$, 3) would be the value that should be restored for $x$. Similarly, if $T_1$ had committed before $T_2$ wrote $x$, then the before image of Write$_2$($x$, 3) would be 2, again the value that should be restored for $x$ if $T_2$ aborts.

We can avoid these problems by requiring that the execution of a Write($x$, *val*) be delayed until all transactions that have previously written $x$ are either committed or aborted. This is similar to the requirement that was needed to avoid cascading aborts. In that case we had to delay all Read($x$) operations until all transactions that had previously written $x$ had either committed or aborted.

Executions that satisfy *both* of these conditions are called *strict*. That is, a DBS that ensures strict executions delays both Reads and Writes for $x$ until all transactions that have previously written $x$ are committed or aborted. Strict executions avoid cascading aborts and are recoverable.

The requirement that executions be recoverable was born out of purely semantic considerations. Unless executions are recoverable, we cannot ensure the integrity of operation semantics. However, pragmatic considerations have led us to require an even stronger condition on the executions, namely, strictness. In this way cascading aborts are eliminated and the Abort operation can be implemented using before images.[2]

## 1.3 SERIALIZABILITY

### Concurrency Control Problems

When two or more transactions execute concurrently, their database operations execute in an *interleaved* fashion. That is, operations from one program may execute in between two operations from another program. This interleaving can cause programs to behave incorrectly, or *interfere*, thereby leading to an inconsistent database. This interference is entirely due to the interleaving. That is, it can occur even if each program is coded correctly and no component of the system fails. The goal of concurrency control is to avoid interference and thereby avoid errors. To understand how programs can interfere with each other, let's look at some examples.

Returning to our banking example, suppose we have a program called Deposit, which deposits money into an account.

```
Procedure Deposit begin
    Start;
    input(account#, amount);
    temp := Read(Accounts[account#]);
    temp := temp + amount;
    Write(Accounts[account#], temp);
    Commit
end
```

Suppose account 13 has a balance of $1000 and customer 1 deposits $100 into account 13 at about the same time that customer 2 deposits $100,000 into account 13. Each customer invokes the Deposit program thereby creating a transaction to perform his or her update. The concurrent execution of these Deposits produces a sequence of Reads and Writes on the database, such as

[2] In [Gray et al. 75], strict executions are called *degree 2 consistent*. *Degree 1 consistency* means that a transaction may not overwrite uncommitted data, although it may read uncommitted data. *Degree 3 consistency* roughly corresponds to serializability, which is the subject of the next section.

Read$_1$(Accounts[13])     returns the value \$1000

Read$_2$(Accounts[13])     returns the value \$1000

Write$_2$(Accounts[13], \$101,000)

Commit$_2$

Write$_1$(Accounts[13], \$1100)

Commit$_1$

The result of this execution is that Accounts[13] contains \$1100. Although customer 2's deposit was successfully accomplished, its interference with customer 1's execution of Deposit caused customer 2's deposit to be lost. This *lost update* phenomenon occurs whenever two transactions, while attempting to modify a data item, both read the item's old value before either of them writes the item's new value.

Another concurrency control problem is illustrated by the following program, called PrintSum, which prints the sum of the balances of two accounts.

```
Procedure PrintSum begin
    Start;
    input(account1, account2);
    temp1 := Read(Accounts[account1]);
    output(temp1);
    temp2 := Read(Accounts[account2]);
    output(temp2);
    temp1 := temp1 + temp2;
    output(temp1);
    Commit
end
```

Suppose accounts 7 and 86 each have a balance of \$200, and customer 3 prints the balances in accounts 7 and 86 (using PrintSum) at about the same time that customer 4 transfers \$100 from account 7 to account 86 (using Transfer, discussed previously under Transaction Syntax). The concurrent execution of these two transactions might lead to the following execution of Reads and Writes.

Read$_4$(Accounts[7])     returns the value \$200

Write$_4$(Accounts[7], \$100)

Read$_3$(Accounts[7])     returns the value \$100

Read$_3$(Accounts[86])     returns the value \$200

Read$_4$(Accounts[86])     returns the value \$200

Write$_4$(Accounts[86], \$300)

Commit$_4$

Commit$_3$

Transfer interferes with PrintSum in this execution, causing PrintSum to print the value $300, which is not the correct sum of balances in accounts 7 and 86. Printsum did not capture the $100 in transit from account 7 to 86. Notice that despite the interference, Transfer still installs the correct values in the database.

This type of interference is called an *inconsistent retrieval.* It occurs whenever a retrieval transaction reads one data item before another transaction updates it and reads another data item after the same transaction has updated it. That is, the retrieval only sees some of the update transaction's results.

### Serializable Executions

In the preceding examples, the errors were caused by the interleaved execution of operations from different transactions. The examples do not exhaust *all* possible ways that concurrently executing transactions can interfere, but they do illustrate two problems that often arise from interleaving. To avoid these and other problems, the kinds of interleavings between transactions must be controlled.

One way to avoid interference problems is not to allow transactions to be interleaved at all. An execution in which no two transactions are interleaved is called serial. More precisely, an execution is *serial* if, for every pair of transactions, all of the operations of one transaction execute before any of the operations of the other. From a user's perspective, in a serial execution it looks as though transactions are operations that the DBS processes atomically. Serial executions are correct because each transaction individually is correct (by assumption), and transactions that execute serially cannot interfere with each other.

One could require that the DBS actually process transactions serially. However, this would mean that the DBS could not execute transactions concurrently, for concurrency means interleaved executions. Without such concurrency, the system may make poor use of its resources, and so might be too inefficient. Only in the simplest systems is serial execution a practical way to avoid interference.

We can broaden the class of allowable executions to include executions that have the same effect as serial ones. Such executions are called serializable. More precisely, an execution is *serializable* if it produces the same output and has the same effect on the database as some serial execution of the same transactions. Since serial executions are correct, and since each serializable execution has the same effect as a serial execution, serializable executions are correct too.

The executions illustrating lost updates and inconsistent retrievals are not serializable. For example, executing the two Deposit transactions serially, in either order, gives a different result than the interleaved execution that lost an update, so the interleaved execution is not serializable. Similarly, the interleaved execution of Transfer and PrintSum has a different effect than every serial execution of the two transactions, and so is not serializable.

Although these two interleaved executions are not serializable, many others are. For example, consider this interleaved execution of Transfer and PrintSum.

Read$_4$(Accounts[7]) returns the value \$200
Write$_4$(Accounts[7], \$100)
Read$_3$(Accounts[7]) returns the value \$100
Read$_4$(Accounts[86]) returns the value \$200
Write$_4$(Accounts[86], \$300)
Commit$_4$
Read$_3$(Accounts[86]) returns the value \$300
Commit$_3$

This execution has the same effect as serially executing Transfer followed by PrintSum. In such a serial execution, Read$_3$(Accounts[7]) immediately follows Write$_4$(Accounts[86], \$300). Although the order of execution of operations in this serial execution is different from the interleaved execution, the effect of each operation is exactly the same as in the interleaved execution. Thus, the interleaved execution is serializable.

Serializability is the definition of correctness for concurrency control in DBSs. Given the importance of the concept, let us explore its strengths and weaknesses.

Most importantly, a DBS whose executions are serializable is easy to understand. To its users, it looks like a sequential transaction processor. A programmer can therefore write each transaction as if it will execute all by itself on a dedicated machine. Potential interference from other transactions is precluded and hence can be ignored.

A DBS that produces serializable executions avoids the kind of interference illustrated by the earlier examples of lost updates and inconsistent retrievals. A lost update occurs when two transactions both read the old value of a data item and subsequently both update that data item. This cannot happen in a serial execution, because one of the transactions reads the data item value written by the other. Since every serializable execution has the same effect as a serial execution, serializable executions avoid lost updates.

An inconsistent retrieval occurs when a retrieval transaction reads some data items before an update transaction updates them and reads some other data items after the update transaction updates them. This cannot happen in a serial execution, because the retrieval transaction reads all of the data items either before the update transaction performs any updates, or after the update transaction performs all of its updates. Since every serializable execution has the same effect as some serial execution, serializable executions avoid inconsistent retrievals too.

### Consistency Preservation

The concept of consistent retrieval can be generalized to apply to the entire database, not just to the data items retrieved by one transaction. This generalization provides another explanation of the value of serializability.

Assume that some of the states of the database are defined to be *consistent*. The database designer defines *consistency predicates* that evaluate to true for the consistent states and false for the other (*inconsistent*) states. For example, suppose we augment the banking database of Accounts to include a data item, Total, which contains the sum of balances in all accounts. A consistency predicate for this database might be "Total is the sum of balances in Accounts." The database state is consistent if and only if (*iff*) the predicate is true.

As part of transaction correctness, we then require that each transaction *preserve database consistency*. That is, whenever a transaction executes on a database state that is initially consistent, it must leave the database in a consistent state after it terminates. For example, Transfer preserves database consistency, but Deposit does not, because it does not update Total after depositing money into an account. To preserve database consistency, Deposit needs to be modified to update Total appropriately.

Notice that each Write in Transfer, taken by itself, does not preserve database consistency. For example, Write(Accounts[oldaccount], temp − amount) unbalances the accounts temporarily, because after it executes, Accounts and Total are inconsistent. Such inconsistencies are common after a transaction has done some but not all of its Writes. However, as long as a transaction fixes such inconsistencies before it terminates, the overall effect is to preserve consistency, and so the transaction is correct.

Consistency preservation captures the concept of producing database states that are meaningful. If each transaction preserves database consistency, then any serial execution of transactions preserves database consistency. This follows from the fact that each transaction leaves the database in a consistent state for the next transaction. Since every serializable execution has the same effect as some serial execution, serializable executions preserve database consistency too.

### Ordering Transactions

All serializable executions are equally correct. Therefore, the DBS may execute transactions in *any* order, as long as the effect is the same as that of *some* serial order. However, not all serial executions produce the same effect. Sometimes a user may prefer one serial execution of transactions over another. In such a case, it is the *user's* responsibility to ensure that the preferred order actually occurs.

For example, a user may want her Deposit transaction to execute before her Transfer transaction. In such a case, she should not submit the transactions at the same time. If she does, the DBS can execute the transactions' operations in any order (e.g., the Transfer before the Deposit). Rather, she should first submit the Deposit transaction. Only after the system acknowledges that the Deposit transaction is committed should she submit the Transfer transaction. This guarantees that the transactions are executed in the desired order.[3]

We will be constructing schedulers that only guarantee serializability. If users must ensure that transactions execute in a particular order, they must secure that order by mechanisms outside the DBS.

### Limitations of Serializability

In many types of computer applications, serializability is not an appropriate goal for controlling concurrent executions. In fact, the concept of transaction may not even be present. In these applications, methods for attaining serializability are simply not relevant.

For example, a statistical application may be taking averages over large amounts of data that is continually updated. Although inconsistent retrievals may result from some interleavings of Reads and Writes, such inconsistencies may only have a small effect on the calculation of averages, and so may be unimportant. By not controlling the interleavings of Reads and Writes, the DBS can often realize a significant performance benefit — at the expense of serializability.

As another example, process control programs may execute forever, each gathering or analyzing data to control a physical process. Since programs never terminate, serial executions don't make sense. Thus, serializability is not a reasonable goal.

A common goal for concurrency control in systems with nonterminating programs is *mutual exclusion*. Mutual exclusion requires the section of a program that accesses a shared resource to be executed by at most one program at a time. Such a section is called a *critical section*. We can view a critical section as a type of transaction. Mutual exclusion ensures that critical sections (i.e., transactions) that access the same resource execute serially. This is a strong form of serializability.

[3] If two transactions do not interact, then it is possible that the user cannot control their effective order of execution. For example, suppose the user waits for $T_1$ to commit before submitting $T_2$, and suppose no data item is accessed by both transactions. If other transactions were executing concurrently with $T_1$ and $T_2$, it is still possible that the only serial execution equivalent to the interleaved execution that occurred is one in which $T_2$ precedes $T_1$. This is odd, but possibly doesn't matter since $T_1$ and $T_2$ don't interact. However, consider the discussion of Terminal I/O in Section 1.2. If the user uses the output of $T_1$ to construct the input to $T_2$, then $T_1$ *must* effectively execute before $T_2$. This incorrect behavior is prevented by the most popular concurrency control method, two phase locking (see Chapter 3), but not by all methods. This rather subtle point is explored further in Exercises 2.12 and 3.4.

Many techniques have been developed for solving the mutual exclusion problem, including locks, semaphores, and monitors. Given the close relationship between mutual exclusion and serializability, it is not surprising that some mutual exclusion techniques have been adapted for use in attaining serializability. We will see examples of these techniques in later chapters.

## 1.4 DATABASE SYSTEM MODEL

In our study of concurrency control and recovery, we need a model of the internal structure of a DBS. In our model, a DBS consists of four *modules* (see Fig. 1–1): a *transaction manager*, which performs any required preprocessing of database and transaction operations it receives from transactions; a *scheduler*, which controls the relative order in which database and transaction operations are executed; a *recovery manager*, which is responsible for transaction commitment and abortion; and a *cache manager*, which operates directly on the database.[4]

Database and transaction operations issued by a transaction to the DBS are first received by the transaction manager. The operations then move down through the scheduler, recovery manager, and cache manager. Thus, each module sends requests to and receives replies from the next lower level module.

We emphasize that this model of a DBS is an *abstract* model. It does not correspond to the software architecture of any DBS we know of. The modules themselves are often more tightly integrated, and therefore less clearly separable, than the model would suggest. Still, for pedagogical reasons, we believe it is important to cleanly separate concurrency control and recovery from other functions of a DBS. This also makes the model a good tool for thought. In later chapters, we will discuss more realistic software architectures for performing the functions of the model.

For most of this section, we will assume that the DBS executes on a *centralized* computer system. Roughly speaking, this means the system consists of a central processor, some main memory, secondary storage devices (usually disks), and I/O devices. We also consider any multiprocessor configuration in which each processor has direct access to all of main memory and to all I/O devices to be a centralized system. A system with two or more processors that do not have direct access to shared main memory or secondary storage devices is called a *distributed* computer system. We extend our model of a centralized DBS to a distributed environment in the final subsection.

[4] [Gray 78] uses "transaction manager" to describe what we call the scheduler and recovery manager, and "database manager" to describe what we call the transaction manager and cache manager.

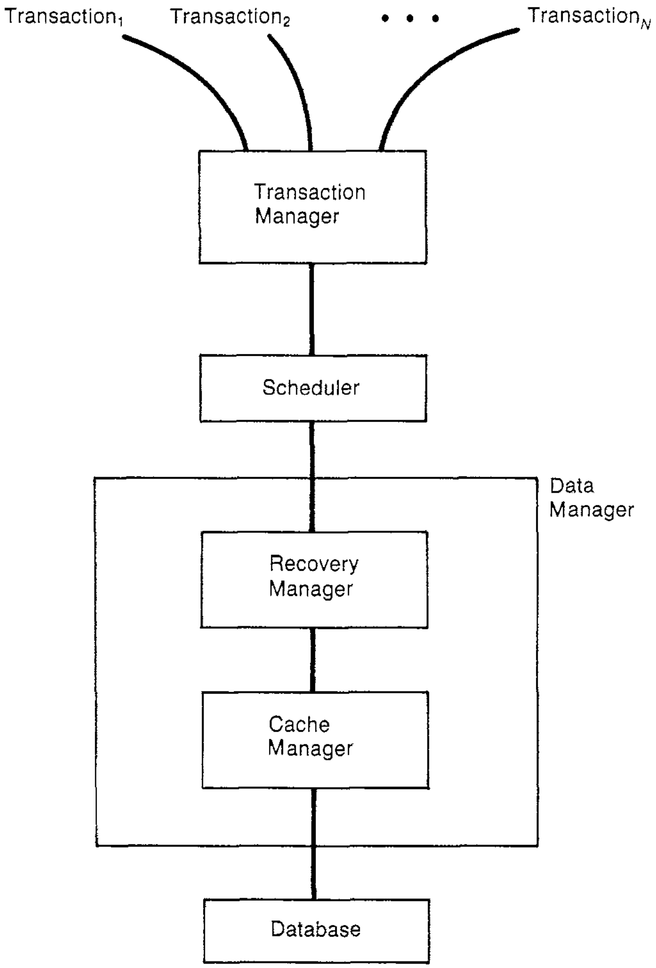

**FIGURE 1–1**
Centralized Database System

### The Cache Manager

A computer system ordinarily offers both volatile and stable storage. *Volatile storage* can be accessed very efficiently, but is susceptible to hardware and operating system failures. Due to its relatively high cost, it is limited in size. *Stable storage* is resistant to failures, but can only be accessed more slowly. Due to its relatively low cost, it is usually plentiful. In today's technology, volatile storage is typically implemented by semiconductor memory and stable storage is implemented by disk devices.

Due to the limited size of volatile storage, the DBS can only keep part of the database in volatile storage at any time. The portion of volatile storage set aside for holding parts of the database is called the *cache*. Managing the cache is the job of the *cache manager* (*CM*). The CM moves data between volatile and stable storage in response to requests from higher layers of the DBS.

Specifically, the CM supports operations Fetch($x$) and Flush($x$). To process Fetch($x$), the CM retrieves $x$ from stable storage into volatile storage. To process Flush($x$), the CM transfers the copy of $x$ from volatile storage into stable storage.

There are times when the CM is unable to process a Fetch($x$) because there is no space in volatile storage for $x$. To solve this problem, the CM must make room by flushing some other data item from volatile storage. Thus, in addition to supporting the Flush operation for higher levels of the DBS, the CM sometimes executes a Flush for its own purposes.

### The Recovery Manager

The *recovery manager* (*RM*) is primarily responsible for ensuring that the database contains all of the effects of committed transactions and none of the effects of aborted ones. It supports the operations Start, Commit, Abort, Read, and Write. It processes these operations by using the Fetch and Flush operations of the CM.

The RM is normally designed to be resilient to failures in which the entire contents of volatile memory are lost. Such failures are called *system failures*. After the computer system recovers from a system failure, the RM must ensure that the database contains the effects of all committed transactions and no effects of transactions that were aborted or active at the time of the failure. It should eliminate the effects of transactions that were active at the time of failure, because those transactions lost their internal states due to the loss of main memory's contents and therefore cannot finish executing and commit.

After a system failure, the only information the RM has available is the contents of stable storage. Since the RM never knows when a system failure might occur, it must be very careful about moving data between volatile and stable storage. Otherwise, it may be caught after a system failure in one of two unrecoverable situations: (1) stable storage does not contain an update by some committed transaction, or (2) stable storage contains the value of $x$ written by some uncommitted transaction, but does not contain the last value of $x$ that was written by a committed transaction. To avoid these problems, the RM may need to restrict the situations in which the CM can unilaterally decide to execute a Flush.

The RM may also be designed to be resilient to failures of portions of stable storage, called *media failures*. To do this, it needs to keep redundant copies of data on at least two different stable storage devices that are unlikely to fail at the same time. To cope with media failures, it again needs to be able to return the database to a state that contains all of the updates of committed transactions and none of the updates of uncommitted ones.

It will frequently be useful to deal with the RM and CM as if it were a single module. We use the term *data manager* (*DM*) to denote that module. The interface to this module is exactly that of the RM. That is, CM functions are hidden from higher levels.

### Schedulers

A scheduler is a program or collection of programs that controls the concurrent execution of transactions. It exercises this control by restricting the order in which the DM executes Reads, Writes, Commits, and Aborts of different transactions. Its goal is to order these operations so that the resulting execution is serializable and recoverable. It may also ensure that the execution avoids cascading aborts or is strict.

To execute a database operation, a transaction passes that operation to the scheduler. After receiving the operation, the scheduler can take one of three actions:

*1.* Execute: It can pass the operation to the DM. When the DM finishes executing the operation, it informs the scheduler. Moreover, if the operation is a Read, the DM returns the value(s) it read, which the scheduler relays back to the transaction.

*2.* Reject: It can refuse to process the operation, in which case it tells the transaction that its operation has been rejected. This causes the transaction to abort. The Abort can be issued by the transaction or by the transaction manager.

*3.* Delay: It can delay the operation by placing it in a queue internal to the scheduler. Later, it can remove the operation from the queue and either execute it or reject it. In the interim (while the operation is being delayed), the scheduler is free to schedule other operations.

Using its three basic actions — executing an operation, rejecting it, or delaying it — the scheduler can control the order in which operations are executed. When it receives an operation from the transaction, it usually tries to pass it to the DM right away, if it can do so without producing a nonserializable execution. If it decides that executing the operation may produce an incorrect result, then it either delays the operation (if it may be able to correctly process the operation in the future) or reject the operation (if it will never be able to correctly process the operation in the future). Thus, it uses execution, delay, and rejection of operations to help produce correct executions.

For example, let's reconsider from the last section the concurrent execution of two Deposit transactions, which deposit \$100 and \$100,000 into account 13:

$\text{Read}_1$(Accounts[13]);<br>
$\text{Read}_2$(Accounts[13]);<br>
$\text{Write}_2$(Accounts[13], \$101,000);<br>
$\text{Commit}_2$;<br>
$\text{Write}_1$(Accounts[13], \$1100);<br>
$\text{Commit}_1$.

To avoid this nonserializable execution, a scheduler might decide to reject $\text{Write}_1$, thereby causing transaction $T_1$ to abort. In this case, the user or transaction manager can resubmit $T_1$, which can now execute without interfering with $T_2$. Alternatively, the scheduler could prevent the above execution by delaying $\text{Read}_2$ until after it receives and processes $\text{Write}_1$. By delaying $\text{Read}_2$, it avoids having to reject $\text{Write}_1$ later on.

The scheduler is quite limited in the information it can use to decide when to execute each operation. We assume that it can only use the information that it obtains from the operations that transactions submit. The scheduler does *not* know any details about the programs comprising the transactions, except as conveyed to it by operations. It can predict neither the operations that will be submitted in the future nor the relative order in which these operations will be submitted. When this type of advance knowledge about programs or operations is needed to make good scheduling decisions, the transactions must explicitly supply this information to the scheduler via additional operations. Unless stated otherwise, we assume such information is not available.

The study of concurrency control techniques is the study of scheduler algorithms that attain serializability and either recoverability, cascadelessness, or strictness. Most of this book is devoted to the design of such algorithms.

### Transaction Manager

Transactions interact with the DBS through a *transaction manager* (*TM*). The TM receives database and transaction operations issued by transactions and forwards them to the scheduler. Depending on the specific concurrency control and recovery algorithms that are used, the TM may also perform other functions. For example, in a distributed DBS the TM is responsible for determining which site should process each operation submitted by a transaction. We'll discuss this more in a moment.

### Ordering Operations

Much of the activity of concurrency control and recovery is ensuring that operations are executed in a certain order. It is important that we be clear and precise about the order in which each module processes the operations that are presented to it. In the following discussion, we use the generic term *module* to describe any of the four DBS components: TM, scheduler, RM, or CM.

At any time, a module is allowed to execute *any* of the unexecuted operations that have been submitted to it. For example, even if the scheduler submits operation $p$ to the RM before operation $q$, the RM is allowed to execute $q$ before $p$.

When a module wants two operations to execute in a particular order, it is the job of the module that *issues* the operations to ensure that the desired order is enforced. For example, if the scheduler wants $p$ to execute before $q$, then it should first pass $p$ to the RM and wait for the RM to acknowledge $p$'s execution; after the acknowledgment, it can pass $q$, thereby guaranteeing that $p$ executes before $q$. This sequence of events — pass an operation, wait for an acknowledgment, pass another operation — is called a *handshake*. We assume that each module uses handshaking whenever it wants to control the order in which another module executes the operations it submits.

As an alternative to handshaking, one could enforce the order of execution of operations by having modules communicate through *first-in–first-out queues*. Each module receives operations from its input queue in the same order that the operations were placed in the queue, and each module is required to process operations in the order they are received. For example, if the CM were to use an input queue, then the RM could force the CM to execute $p$ before $q$ by placing $p$ in the queue before $q$.

We do not use queues for intermodule communication for two reasons. First, they unnecessarily force a module to process operations strictly sequentially. For example, even if the RM doesn't care in what order $p$ and $q$ are executed, by placing them in the CM queue it forces the CM to process them in a particular order. In our model, if the RM doesn't care in which order $p$ and $q$ are processed, then it would pass $p$ and $q$ without handshaking, so the CM could process the operations in either order.

Second, when three or more modules are involved in processing operations, queues may not be powerful enough to enforce orders of operations. For example, suppose two modules perform the function of data manager, say $DM_1$ and $DM_2$. ($DM_1$ and $DM_2$ might be at different sites of a distributed system.) And suppose the scheduler wants $DM_1$ to process $p$ before $DM_2$ processes $q$. The scheduler can enforce this order using handshaking, but not using queues. Even if $DM_1$ and $DM_2$ share an input queue, they need a handshake to ensure the desired order of operations.

*Except when we explicitly state otherwise, we assume that handshaking is used for enforcing the order of execution of operations.*

### Distributed Database System Architecture

A *distributed database system* (or *distributed DBS*) is a collection of *sites* connected by a communication network (see Fig. 1–2). We assume that two

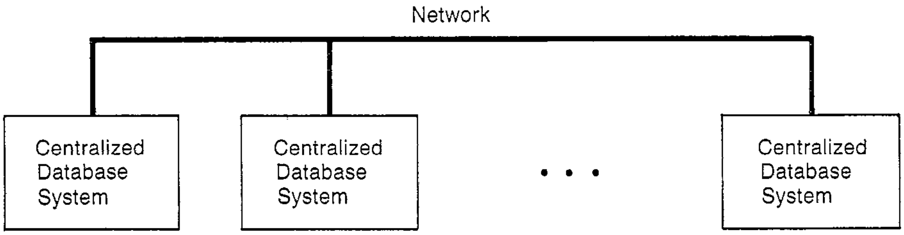
Network
Centralized Database System
Centralized Database System
• • •
Centralized Database System

**FIGURE 1–2**
Distributed Database System

processes can exchange messages whether they are located at the same site or at different sites (in which case the messages are sent over the communication network).

Each site is a centralized DBS, which stores a portion of the database. We assume that each data item is stored at exactly one site.[5] Each transaction consists of one or more processes that execute at one or more sites. We assume that a transaction issues each of its operations to whichever TM is most convenient (e.g., the closest). When a TM receives a transaction's Read or Write that cannot be serviced at its site, the TM forwards that operation to the scheduler at another site that has the data needed to process the operation. Thus, each TM can communicate with every scheduler by sending messages over the network.

## BIBLIOGRAPHIC NOTES

Research publications on transaction management began appearing in the early to mid 1970s [Bjork 72, Davies 72], [Bjork 73] [Chamberlin, Boyce, Traiger 74], and [Davies 73], although the problem was probably studied even earlier by designers of the first on-line systems in the 1960s. By 1976, it was an active research area with a steady stream of papers appearing. Some of the early influential ones include [Eswaran et al. 76], [Gray et al. 75], and [Stearns, Lewis, Rosenkrantz 76].

Concurrency control problems had been treated in the context of operating systems beginning in the mid 1960s. [Ben-Ari 82], [Brinch Hansen 73], and [Holt et al. 78] survey this work, as do most textbooks on operating systems.

Recovery was first treated in the context of fault-tolerant hardware design, and later in general purpose program design. Elements of the transaction concept appeared in the

[5] In Chapter 8, on replicated data, we will allow a data item to be stored at multiple sites.

"recovery block" proposal of [Horning et al. 74]. Atomic actions (transactions) in this context were proposed in [Lomet 77b]. Surveys of hardware and software approaches to fault tolerance appear in [Anderson, Lee 81], [Shrivastava 85], and [Siewiorek 82].

An interesting extension of the transaction abstraction is to allow transactions to be nested as subtransactions within larger ones. Several forms of nested transactions have been implemented [Gray 81], [Liskov, Scheifler 83], [Moss 85], [Mueller, Moore, Popek 83], and [Reed 78]. Theoretical aspects of nested transactions are described in [Beeri et al. 83], [Lynch 83b], and [Moss, Griffeth, Graham 86]. We do not cover nested transactions in this book.

## EXERCISES

1.1 For each of the example executions in Section 1.2, determine if it is serializable, assuming each active transaction ultimately commits.

1.2 Explain why each example execution in Section 1.3 is or is not recoverable, cascadeless, or strict.

1.3 Suppose transaction $T_1$ reads $x$, then reads $y$, then writes $x$, and then writes $y$. Suppose $T_2$ reads $y$ and then writes $x$. Give example executions of $T_1$ and $T_2$ that are serializable and

a. recoverable but not cascadeless;

b. cascadeless but not strict; and

c. strict.

Now, give example executions that are *not* serializable and satisfy (a), (b), and (c).

1.4 We assumed that transactions only interact through their accesses to the database. We can weaken this assumption slightly by allowing transactions to exchange messages that are not part of the database in the following case: A transaction $T_1$ can receive a message from transaction $T_1$ provided that the DBS processed $T_1$'s Commit before it processed $T_2$'s Read of $T_1$'s message. Explain why this weakened assumption is still satisfactory by analyzing its effects on recoverability and serializability.

1.5 Using the banking database of this chapter, write a program that takes two account numbers as input, determines which account has the larger balance, and replaces the balance of the smaller account by that of the larger. What are the possible sequences of Reads and Writes that your program can issue?

1.6 Give an example program for the banking application that, when executed as a transaction, has terminal output that cannot be deferred.