This page is intentionally left blank.

Chapter 11

Methods to Gain Reliability

As far as we know, our computer has never had an undetected error.

Conrad H. Weisert
Union Carbide Corporation

This, and the two chapters which follow, discuss the areas that, in combination, are concerned with the security of database systems. The term *security* is used here to describe the protection of systems and their contents from destruction. The term security has often been used in a more narrow sense, but with a scope that varies from discussion to discussion. We will not cover the aspects of security which concern themselves with issues of building and personnel security, although it is obvious that without physical security most computer-system oriented efforts can be brought to naught.

If we wish to secure a database, we will need to achieve a reliable and predictable mode of operation; we also have to provide a protection mechanism to achieve the desired control of access to the data. Finally, we have to assure that there will be no destructive interference as multiple users share access to the database.

These three issues of security are presented in Chaps. 11, 12, and 13:

Reliability: improving the probability that the system does what it is instructed to do.

Protection: understanding, organizing, and controlling access to data in accordance with specified rights.

Integrity: maintenance of security in a system where multiple users are permitted to access the system and share the database.

There is clearly a strong interrelation among these areas. The partitioning is chosen to simplify the presentation. Some general issues on cost and benefits related to security problems will be discussed in Chap. 12 on the protection of data.

11-1 RELIABILITY

Reliability is achieved when a computer system, both hardware and software, always produces correct results. One obvious problem is the determination of correctness. We use the term *fault* to denote the cause, and the word *error* for the manifestation of a fault. Lack of correctness is demonstrated by the occurrence of an error.

Hardware, database design, programs, and data input mechanisms all contain faults. Program faults or *bugs* are familiar to us all.

Current methods of program debugging by testing are clearly inadequate, since they are typically limited to the verification of a few sample computations. Analysis of the program structure can create a test plan which assures that each program section is executed once, but testing of every possible execution sequence is rarely feasible, since the number of combinations is nearly always too large. Extreme values of input data can often induce errors and are useful in testing.

The application of structured programming techniques should lead to a reduction and better identification of *bugs* but will still not eliminate them. Formal verification of programs attempts to eliminate faults. These techniques depend on a detailed specification of the data transforms, and here a comprehensive database model is essential.

We will not discuss the important specific issue of program faults further but will concentrate on methods to deal with all kinds of faults found in databases. Procedures of integrity monitoring (Sec. 13-3) can also help in the maintenance of database reliability.

In order to produce correct results, we need correct data and correct algorithms, and the system has to carry out the algorithms correctly. In each of these areas the problem reduces to two subproblems:

1 The existence of a fault has to be detected, or the absence of faults has to be proved.

2 When an error is detected, a means of correcting and recovering from the error has to be available.

Frequently, a single technique may provide both detection and also some restoration capability. We will discuss some common techniques in the light of these two aspects and then evaluate how these methods may be combined into a system that will have desirable characteristics.

Failure Probabilities Computer systems are composed of many parts that are prone to failure, so that success in a given time period is achieved only if none of the parts fails. For q parts having each a failure probability of p_f the probability of successful operation is

$$p_{success} = (1 - p_f)^q \qquad 11\text{-}1$$

We will look at the effects of failures using some simple examples.

If 10 parts, each with a probability of failure of 1% for a given operation, are used in a logically serial arrangement, the probability of achieving the correct result is

$$p_{success} = 0.99^{10} = 0.904$$

which amounts to a failure probability of nearly 10%.

For 100 such parts, the probability of system operation decreases to

$$p_{success} = 0.99^{100} = 37\%$$

It should be noted that parallel electric circuits still have serial failure characteristics. If one bit out of a 16-bit word is not transmitted correctly between storage and a device controller, the entire result is wrong.

MTBF Computer-system components have very high reliability probabilities. The failure rate of modern electronics is not greatly affected by the number of operations carried out, and hence is specified in terms of operational time. A typical *mean time between failures* (MTBF) for a highly loaded digital switching transistor is 40 000 hours. This time is equivalent to an error probability of only 0.000 025 per hour. A moderately large system containing and using 10 000 such components would, however, have a probability of only 0.78 of avoiding an error in an hour's span or a MTBF of 1.3 hours. The error rate of components is significantly reduced by conservative usage and a controlled environment. In practice, not all component errors will have a detectable effect in terms of the computer's results, since not all components are contributing to the operation of the computer at any one time, and hence are not actually in use. On highly loaded systems failure rates do increase.

Storage-System Failures Mechanical parts of computer systems have higher error rates than electronics. Magnetic storage devices, communication lines, and data entry and output devices are quite prone to errors. Human beings are essential elements of computer systems and show even higher error rates.

We can make, however, one important observation about modern disk drives. *A block will be written either completely or not at all.* The electrical and mechanical inertia in these systems is typically sufficient to allow a write operation, once started, to complete completely and correctly. This means that, if blocks are properly managed, a high level of system reliability can be achieved.

Failure Control It is important to realize the continued existence of faults and to provide resources to cope with errors as well as to prevent errors. Early detection is needed to stop propagation of errors. For types of errors which occur frequently,

automatic correction or recovery is required. Since the same error can be due to more than one fault, the proper recovery procedure may not be obvious. Halting the entire system whenever errors occur can have a very high cost not only in terms of system unavailability but also in terms of confusion generated if the cause for a halt is not clear to everyone affected. In a chaotic environment, fixes and temporary patches are apt to endanger the database more than a reasonable and consistent error recovery algorithm. The formalization of error recovery also enables the application of lessons learned during failure situations, so that the fraction of cases for which the error recovery procedure is correct increases over time.

MTTR The time required to get a system going again is referred to as the *mean time to repair* (MTTR). In this context repair can range from complete fault identification and correction to merely logging of the error occurrence and restarting of the system. The MTTR can be reduced by having a duplicated computer system. This means that more than twice the number of components have to be maintained, since additional components will be needed to connect the two systems. The trade-off between decreased MTBF due to system complexity and increased MTTR has to be carefully evaluated in any system where a high availability of the database is desired.

Use of Replicated Systems A duplicate computer facility can be used fully in parallel, or it can be used to provide backup only, and process less critical work up to the time a failure occurs. In the first case, the MTTR can be as small as the time that it takes to detect the error condition and disconnect the faulty computer.

In the second case, programs running in the backup computer will have to be discontinued, and the processing programs from the failing machine will have to be initiated at a proper point. Here a larger MTTR can be expected; the MTTR in fact may be greater than the time it takes to reinitiate processing on the primary machine if the failures detected were transient. A failure type analysis will be required to decide which of these two alternatives will be best. Making the decision itself may add to the MTTR.

With two machines which operate in parallel, recognition of a discrepancy of results detects the error but does not allow the system to determine automatically which of the duplicated units caused the error. Replication of systems or subsystems in triplicate, *triple modular redundancy* (TMR), has been used in unusually critical applications. This approach allows two computers to annul the output of the failing computer.

The space shuttle system uses four computers, so that TMR can be maintained even after a computer has failed. A fifth one is available as a spare.

Since the cost of processors is dropping rapidly, replication of processing units is becoming more common, and some "*non-stop*" transaction systems are now being delivered with a minimum of 2 and up to 16 processors.

In practice many computer circuit failures are transient and will not repeat for a long time. Such errors may be due to a rare combination of the state of the system, power fluctuations, electrical noise, or accumulation of static electricity.

Hence a *retry* capability, replication of a process along the time axis, can provide many of the same benefits.

In the remainder of this chapter, we will avoid the discussion of complete system replication, since this is a problem which involves the architecture of the entire computer hardware and operating systems. Selective replication, however, can be employed in various ways by file and database systems. Typical hardware replication is the provision of extra disk units, tape drives, or controllers.

Availability The effect of dealing with faults as seen by the user is often measured as the *availability*, namely the fraction of time that the system is capable of production.

$$\text{Availability} = 1 - \frac{\text{time for scheduled maintenance}}{\text{scheduled maintenance interval}} - \frac{\mathsf{MTTR}}{\mathsf{MTBF}} \qquad 11\text{-}2$$

The availability measure does not include the aftereffects of failures to the user. Whenever a failure occurs, the current transaction is lost. In an environment with long transactions the cost of recovery increases substantially with a poor MTBF. If the users depend greatly on the system for running their business, a poor MTTR will be unacceptable.

Recovery Components Those components which are used only to provide recovery capability should be isolated so that their failures do not affect the productive operation of the entire system. Isolation will increase the net MTBF. Errors in recovery components should, however, generate a warning signal, so that repair action can be undertaken. The repair response should be carefully and formally specified, taking into account the possible effects due to insufficient backup during the repair period. Where file integrity is more important than system availability, an immediate controlled shutdown may be in order. There is a tendency by computer operations staff to continue the provision of services, which they view as their major task, rather than initiate repair procedures to eliminate faults that seem not to affect current system operations.

11-2 REDUNDANCY

Redundancy is obtained when data units (bytes, words, blocks, etc.) are expanded to provide more information than is strictly necessary. The data units can be checked for internal consistency. If they are found to be in error they may be corrected.

11-2-1 Parity

A simple form of redundancy is obtained by adding a parity bit to elemental data units. Characters on tape or in computer memory frequently have a parity bit added when they are initially generated. The count of the number of bits of value 1 in the character representation is termed the *Hamming weight*. If the character is represented by 8 bits, the parity bit in the ninth position will be set so that the Hamming weight of the combination is odd (odd parity encoding) or even (even parity encoding). Odd parity is the preferred code, since it avoids any occurrences of all 0 or all blank sections on the recording medium, which is useful to assure

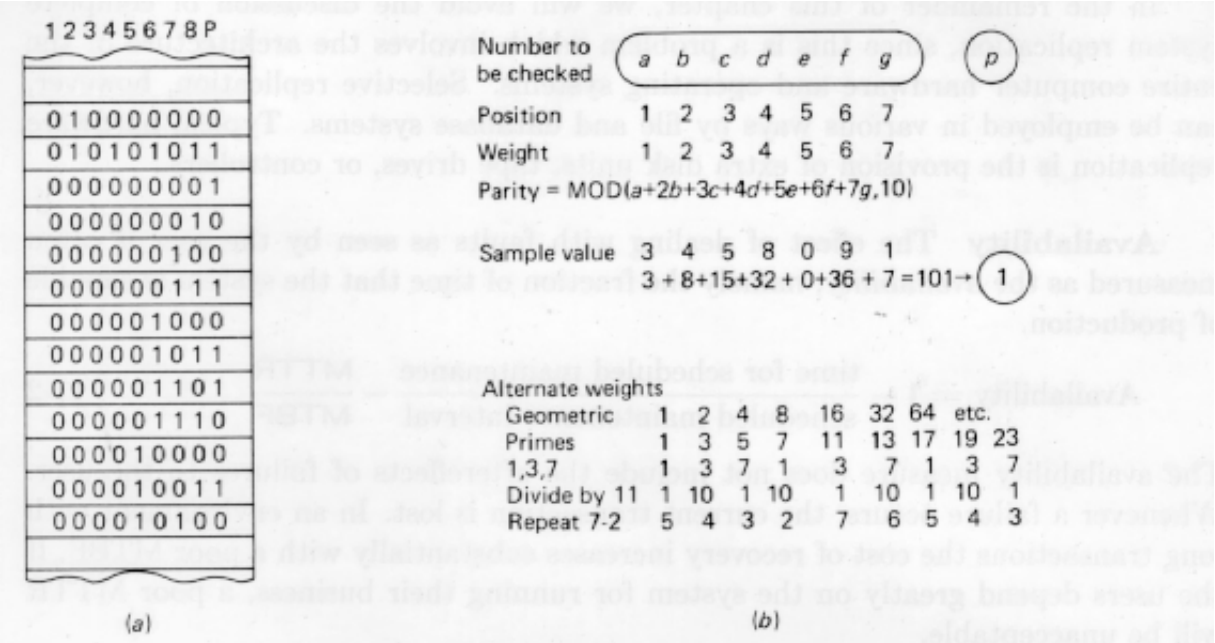

Figure 11-1 Parity. (a) Section of tape with odd parity. (b) Check-digit computation.

detection of the existence of a character, especially on tapes. Figure 11-1*a* shows odd-parity encoding for some 8-bit characters on a 9-track tape.

Odd parity produces a code with a minimum Hamming weight of 1. The counting of bits is done easily by computer hardware, and some machines have specific instructions for bit counting. Most machines perform the encoding and checking processes, until an error occurs, without user intervention.

The addition of the one parity bit doubles the number of possible symbols that a code can transmit. Half of these codes are not valid characters and should not occur in the received input. Their appearance hence signals an error condition. The number of bits which are different between the intended and actual code received is termed the *Hamming distance* of the two codes. The Hamming distance is, of course, equal to the difference in the Hamming weights of the two codes.

Simple parity coding is well suited to the communication-oriented processes in computing. Parity coding is independent of the content of the data. This is both an advantage, in that it can be universally applied, and a disadvantage, since it cannot detect errors related to the content of the data. Techniques similar to parity encoding are also used for numbers which are transmitted outside of computer systems. Decimal identification numbers may have check digits appended which are verifiable. Since accidental transposition of digits is a frequent error, the check digit is best computed not as a simple sum but as a sum of products as shown in Fig. 11-1*b*. Only a one-digit remainder of the sum is retained for checking. An interchange of 3 and 4 in the sample will generate a check digit $p = 0$ instead of $p = 1$. Many systems invert the check digit to simplify checking, so that $p = 1$ becomes 9.

Modulo 10, as used in Fig. 11-1*b*, will not detect certain transposition errors between alternate columns; in the sample a transposition of 5 and 0 will not be detected. Because of the high frequency of transposition errors, use of modulo 9 or 11 ($p = 10 \rightarrow 0$) is preferred even though it reduces the power of the check

digit. Digits in the position equal to the modulus are not checked at all using the simple successive arithmetic weights scheme. Alternate weight assignments which have been used are also shown in Fig. 11-1*b*; the series $(1, 10, 1, \ldots)$ is obtained in effect by division of the number by 11.

Only detection is provided by parity encoding, and even the detection capability is limited. Whereas with simple parity all one-bit errors will be detected, reversals of an even number of bits within a single character will not be noted. In situations where errors are apt to occur in batches or bursts, the detection probability remains high since a number of sequential characters will be affected, many of which will have a detectable parity error.

11-2-2 Duplication

Duplication of data is another simple form of achieving redundancy. It is used during input, in some tape systems, and in the storage of critical data. If the data are processed through different channels and then matched, the error-detection probability will be quite high. To decide which copy to use for correction requires another indicator, for instance, a parity error. This approach is hence mainly effective for hardware faults, program faults will create the same error in both paths. The cost of duplicate entry and maintenance of data is, of course, quite high. Keypunching with subsequent verification of the punched cards by retyping and matching is an example of duplication of data during one small interval of the information-processing cycle. The original document must remain available to arbitrate when an error is found.

When information is copied or transmitted, a duplicate will exist for some length of time. It can be useful to design the system so that this duplicate remains available until correctness of the copy has been verified. The detection of an error can be used to initiate recopying of the data. This technique is prevalent in the transmission of data over transmission lines. Not only is the vulnerability of data during transmission high, but the size of the damaged areas is frequently large. A reason for errors to occur in bursts is that the data are transmitted serially, using separate clocks, one on each end, which are used to define the individual bit positions. A relatively long time will be required to resynchronize these clocks after a timing error. Burst errors are also caused by transients induced when lines are being switched in the dial-telephone network.

Maintenance of completely duplicate files on disks is rarely feasible on general-purpose computers. The cost of completely duplicated storage and of extra channel capacity may also be excessive if one considers the current high reliability of computer storage devices. Complete data redundancy also increases the time required for file update. The computer systems designed for *non-stop* operation have fully duplicated channels and controllers to avoid delays in writing to duplicated or mirrored disk units.

To achieve much of the reliability that a duplicated system can provide, a small amount of selected critical information may be replicated. In a file system, this may be the index information or the linkage between records. The damage due to loss of a data element can be expected to be minor compared with the effect of the loss

of a pointer which causes all data beyond a certain point to become unavailable. Selective replication will be combined with other error-detection mechanisms, perhaps parity, so that it can be determined which copy is the correct one. Then the erroneous file may be restored. In case of doubt the data file contents, rather than pointer information, will be used for correction.

In order to avoid some of the overhead that would be caused by writing indexes or pointer values into duplicate blocks, a copy of the information required to build the linkages may be written as a preface to the data records themselves. This practice avoids additional seek and rotational overhead times and only adds to the record transmission time and storage cost. Restoration of damaged indexes is now more expensive, since it involves analysis of the bulk of the data files. It is hoped that such a recovery action is an infrequent occurrence.

11-2-3 Error-Correcting Codes

Error-correcting codes provide a high degree of the benefits obtained by replication of data without all the costs. Here a certain number of bits are added to each data element. These check bits are produced similarly to the parity bits, but each check bit represents different groupings of the information bits. These groups are organized in such a fashion that on error an indication of which bit is in error will result. The principle is shown in Fig. 11-2.

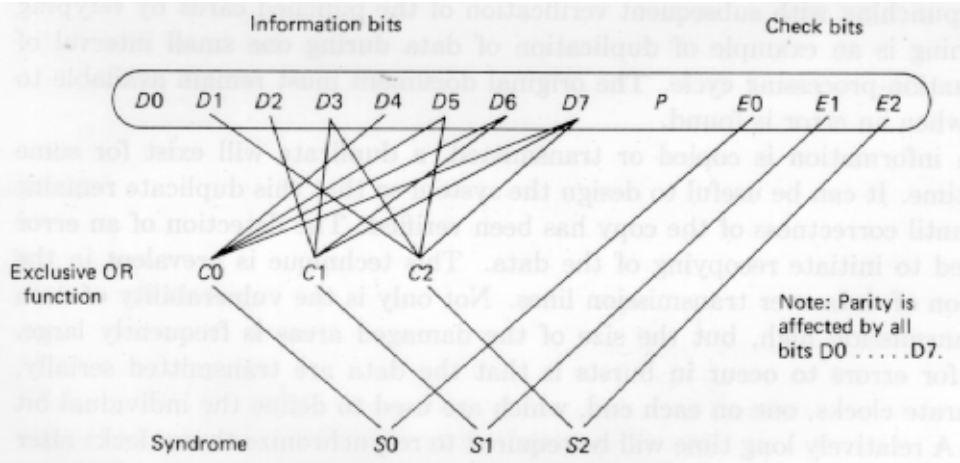

Figure 11-2 The idea behind error correction.

If the parity bit P indicates that there is an error, the three error-code bits C0, C1, C2 can be generated from the data bits and compared with the checkbits E0, E1, E2 that were carried along. A difference in these bits is obtained in the syndrome S, and this will indicate (in binary) the number of the bit in error. For example, an error in D5 will cause a *syndrome* of 101, implicating data bit 5.

This example is incomplete, since it does not provide error control for the parity and the error-correction bits themselves. The number of check bits r required for codes which can correct ec bits and can definitely detect errors of up to ed bits is determined by a minimum Hamming weight of

$$Hw_{min} = 2ec + ed + 1 \qquad 11\text{-}3$$

for the complete error-correcting data codes.

If a frame of coded information has a total of n bits it can represent of 2^n codes. For the byte with 8 data bits in Fig. 11-2 the value of $n = d + r = 12$. We can now consider that out of the 2^n codes are received, only 2^d codes are correct. Out of the remainder $(2^n - 2^d)$ those codes which have a low ($ec = 1, 2,$ ...) Hamming distance to any of the correct codes are used for correction by being converted to their neighbors. The correct codes have to be seperated by a Hamming distance of at least $2ec$. The remainder of the codes is undecodable, and hence will detect, but not correct an error.

Procedures are available to select a code transform matrix. which will generate the least number of check bits r, from the information bits d, given ec, ed. From the transformation matrix, a check matrix can be obtained which will generate a 0 if the code received was correct (or possibly had more than ed errors) and which generates on error a bit-string which can be used to correct an incorrect code, if there were not more than ec errors. An example for $n = 7, d = 4, r = 3, ed = 3, ec = 1$ is shown in Fig. 11-3.

It is obvious that r check bits can at most represent $2^r - 1$ possible errors, since the syndrome composed of all zeros indicates all is well. The number of possible errors, on the other hand, is $2^{(r+d)} - 2^d$. Hence we will want to consider only those matrices which are apt to correct the errors we expect most frequently. Hamming codes select those matrices which correct all the errors from the lowest Hamming distance up. Assuring that all errors involving a limited number of bits are checked is appropriate when errors occur independently. For $ec = 1$ and $ed = 2$, it has been shown that the minimum number of check bits required is determined by

$$2^{r-1} \geq r + d > 2^{r-2} \qquad 11\text{-}4$$

We see that for $d = 8$ this capability is obtained with $r = 5$ check bits.

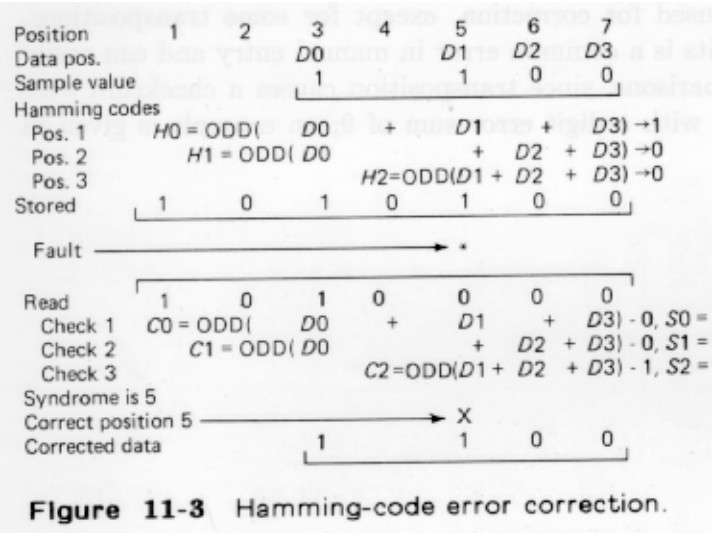

Figure 11-3 Hamming-code error correction.

The number of check bits is increased to detect a larger number of errors and to correct more than single-bit errors. There will be a trade-off when selecting the transformation matrices between the numbers of errors detected and the number of errors correctable. The number of check bits required is a logarithmic function of the

length of the data unit. This fact makes the cost in terms of bits relatively low when the units are long ($d \gg 8$), although the circuitry becomes more complex. In fact, it may be desirable to implement in hardware only the syndrome generation and the check for 0 value of the syndrome, and to call a software procedure for the error correction. Other error-correcting codes, such as Bose-Chaudhuri and Fire codes, provide similar capabilities but are more advantageous for variable-length blocks, since they are cyclic and can correct errors occurring in a data stream rather than only by transformation of a complete block. They also can be designed to cope better with burst-type, that is, dependent bit-errors.

A specific form of a code can correct only up to a finite number of bits, and thus their usefulness is greatest when the prevalent error pattern is known. On tape, for instance, the distance between tracks is much greater than the distance between bits along the tape. An error spot will hence cause damage in a track rather than across tracks, and the codes used for tape take advantage of this fact.

The correction of the error by hardware will delay transmission by the length of the error-correcting cycle. However, since a longer cycle uses relatively fewer check bits, an increase in net transmission rate will occur. Advertised data-transmission rates generally include only the net information bits, so that the actual disk-to-controller bit transfer occurs at a higher rate. The difference between the data and the real transfer rate can be great; the 6250-bpi tapes and mass-storage units described in Sec. 2-1-1 have actual bit densities of approximately 9000 bpi.

Occasional but large bursts of errors found in communication lines make the use of error-correcting codes in data transmission less profitable. Here detection and retransmit techniques are prevalent. Files, however, are maintained on closely coupled devices with controllers which have the logical capability to generate error codes as well as to check and correct errors. Error correction here permits a drastic increase in storage density.

11-2-4 Batch Checking

Where data are manually processed before submission to an input transaction or where additional software checking is desired, *checksums* or *batch totals* may be employed. A checksum is the sum of all values in a critical column. Checksums are typically computed before data entry and within the transaction, and these sums are compared. This method is conceptually identical to parity checking, but since the result is a large number rather than a single bit, the error-detection probability is quite high. For a large batch the added data quantity remains relatively low.

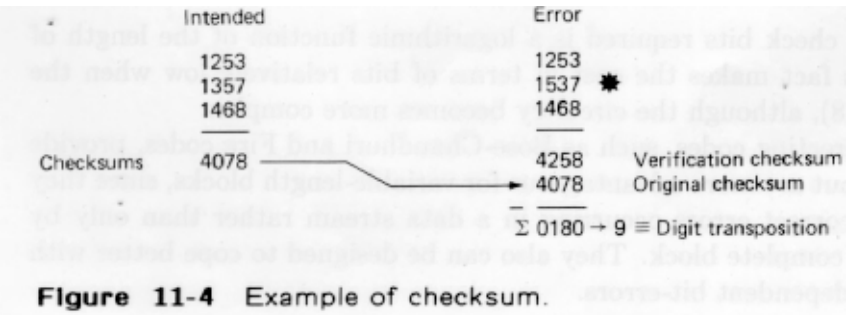

Figure 11-4 Example of checksum.

The method cannot be used for correction, except for some transpositions. Transposition of adjacent digits is a common error in manual entry and can easily be spotted in checksum comparisons, since transposition causes a checksum with errors in two adjacent digits, with a digit error sum of 9; an example is given in Fig. 11-4.

Another technique to avoid errors in manually prepared data is the use of *transaction sequence numbers*. Each batch of input documents for one transaction is assigned a sequential identifier and this number is also entered. The data-entry program can check that all transactions have been received. If it is important that transactions are processed in the same sequence, the check can simply verify that transaction numbers increase by one. Many such systems check transactions in this manner, but the sequentiality constraint may place unnecessary constraints on operations. If one batch is late, all processing ceases.

11-2-5 Natural Redundancy

Even without explicit addition of redundant information, we find that much data coding is highly redundant. A prime example is information coded in English or any other natural language. If the data processed is English text, a dictionary look-up can be used to detect errors, and with intelligent processing, as done by human beings when reading text with typographical errors, a large fraction of single-character errors might corrected.

Frequently the number of choices in coded data are much more limited than the whole of the English language, and simple searches, possibly to lexicons accessed directly or via indexes, can determine whether an entry is valid. Such look-ups are performed routinely to verify items entered such as employee names, identification codes, and similar data elements in many processing systems.

There frequently is a considerable amount of redundancy within the character encoding itself; and this too can be used for error detection. An 8-bit code, for instance, allows 256 characters, but frequently the data are restricted to uppercase letters (26) or alphanumeric (36) characters (see Sec. 14-2). Filtering to detect erroneous characters can be done at little incremental effort during other text-processing operations and can prevent serious problems which may be caused when wrong characters enter communications channels. A bad character may have unexpected control functions such as mode shifts, disconnect, or screen erase on displays.

11-2-6 Error Compensation

Careful system design can reduce the effect of some errors. As an example we will cite a case where data is acquired periodically.

The electric meter on a house is read every month. If a transcription error occurs, there may be an overbilling in one month, but next month, if the reading is entered correctly, there will be a compensating underbilling. There is, in fact, a duplicate memory here; one is the meter and the other is the computer file. If the meter were reset after each reading, a transcription error would not be compensated.

Wherever errors are not compensated the database can diverge from the real world. If an inventory is maintained using counts of items sold and items purchased, periodic verification is essential.

11-2-7 Buffer Protection

File data are most vulnerable while they are in primary memory, especially where computer systems are used for many parallel tasks or undergo frequent change.

Checksums taken of a buffer contents may be used to verify that data have not been changed by other than approved procedure. The maintenance and frequent recalculation of checksums can be quite costly.

Another alternative to protect buffers is the use of *barriers*. These consist of a pair of codes which are apt to occur infrequently. The buffer contents itself is not verified, but the barriers are matched at critical times. Such checks will prevent the use of buffers which have been damaged by overruns or overflows from other areas due to loop or limit code failures. Since a great fraction of programming errors are due to limit failures, and since these errors are not easily detected by their perpetrators, the use of check barriers can provide a fair amount of protection at low cost. No correction capability is provided.

A barrier value which is not a defined character code, not a normalized floating-point number, a very large negative integer, and not a computer instruction is depicted in Fig. 11-5.

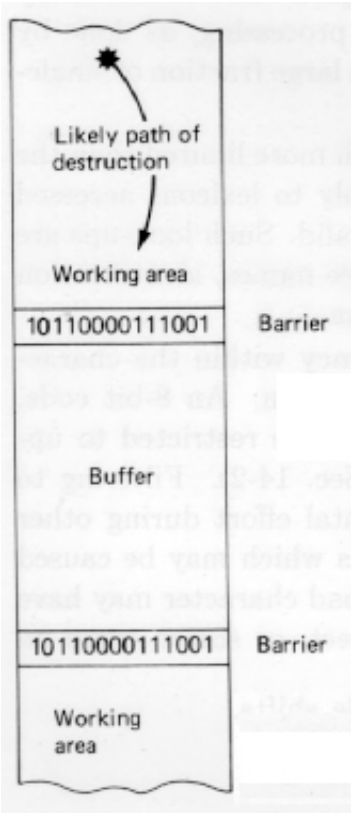

Figure 11-5 Buffer barriers.

11-2-8 Feedback

We discussed in Sec. 5-2 how human error detection can be aided by interactive selection and playback. It cannot be emphasized too much that the best quality control consists of frequent generation of useful output to the various levels of personnel involved in data collection, preparation, and entry. Neither mechanical means nor constant exhortations will keep the contents of a database at an acceptable level when the data are not used regularly.

It is especially important to assure correctness of data which will be stored for long periods of time. A failure in a file of historic data is frequently not correctable, since the source material will have turned to some functional equivalent of dust.

11-3 TRANSACTION RELIABILITY

In the previous sections we have treated general aspects of file-system reliability, and we can now consider issues specific to databases. We relate issues of database reliability with the successful completion of transactions. As presented in Sec. 1-7-5 we define a transaction as a program which changes the state of the database so that, if the initial state of the database was correct and the transaction does not introduce an error due to a fault or bad input data, the state of the database is again correct after the transaction has successfully terminated. If the transaction does not terminate properly, none of its effects should be visible in the database.

The issue of input errors is often ignored in papers discussing transaction control. We do consider input errors in the approaches that will be presented.

We now analyze transactions and in Sec. 11-3-2 techniques to deal with errors.

11-3-1 Two-Phase Transactions

A transaction can fail for a number of reasons. It can find that the input data is not correct or does not match the database. The input data can be poorly formatted or incomplete, it can be outside of schema limits, or it can require computations with data values which do not exist in the database.

The transaction can also find that it cannot proceed because of a lack of resources. Perhaps some needed blocks are currently held by another user, as discussed in Sec. 13-2. Finally the computer itself may fail or *crash* during a transaction.

In order to minimize the effect of a failure, the actions in a transaction are best separated into two phases. During the first phase the transaction acquires all the resources needed for its execution, including any blocks it needs to update. It also carries out any specified computations. Any blocks which do not destroy the prior state of the database can be written during the first phase. During the second phase all remaining changes are placed into the database, and eventually the resources held by the transaction are released.

During the first phase the transaction can terminate without having affected the database in a meaningful way. At worst some blocks have been allocated and not put into use.

Upon entering phase two, the transaction is committed to complete. A failure during the second phase, by an internal fault in the transaction or caused externally, by the system, can have affected the database. Dealing with failures in phase two requires a mechanism to complete the transaction so that a consistent state is again obtained. Phase-two failures will be avoided as much as possible, however, in a real world, they cannot be eliminated.

We can now define the point where a transaction can be aborted without a restart operation as the *commit point*. This definition leads to two cases for error recovery within a transaction:

Transaction abort: An error is recognized during the processing of a transaction, prior to the commit point. An abort command is executed and the affected state of the database is rolled back to the state prior to the begin of the transaction.

Transaction restart: An error is recognized during the processing of a transaction, after the commit point. A restart command is executed and the database actions required by the transaction are completed.

At times a third type of action is required:

Transaction undo: An error is found in the database. The cause is traced to a fault in some completed transaction or to an error in the input data for that transaction. An undo transaction command is executed to correct all effects.

Some of the same techniques can be used for either restart or undo, but both cases are much more complex than transaction abort.

In some transactions it can be awkward to define a single commit point. Long and complex transactions may have been *nested* into a hierarchy.

An example can be a travel agent's trip reservation. Each subtransaction, obtaining the airline ticket, the car rental, the hotel booking, etc., can have a definite commit point. An eventual failure, say the unavailability of hotel space, can force an undo of previously committed subtransactions. Delaying the commit of the separate subtransactions can cause a loss of airline reservations while hotel space is being explored.

The complexities of commit protocols increase where the database is distributed over multiple processing nodes. The decision to commit has to be made jointly by the affected nodes. A commit-managing node may give the final go-ahead to commit to all the participating nodes when notifications of being ready-to-commit have been received from every node. Any node which has indicated a readiness to commit has to remain in a state which permits it to abort or complete the transaction until the go-ahead or an abort message has been received from the commit manager. A centralization of commit management is in conflict with the notions leading to distribution. Alternate techniques have been explored but seem to lead either to a relaxation of update consistency over the nodes or to higher management cost in terms of message transmission [GarciaMolina[81]].

We continue now with the case of simple two-phase transactions.

11-3-2 Transaction Management

In order to deal with the management of transactions we expect the operating system to include a program module called the *transaction manager*. If the operating system does not include adequate transaction services such a module may have to be supplied by the database management system. The transaction manager will receive the following control messages during the execution of a transaction:

```
Transaction begin
Transaction commit  or  Transaction abort
Transaction done
```

Each message will be acknowledged by the transaction manager. The transaction manager will also receive data from the transaction to aid in recovery, if needed.

Abort The transaction program can initiate an abort upon finding an error. The only requirement is to inform the user and the system. The system should

be notified, since it may hold resources for the transaction, which should now be released. A system may also maintain logging information to aid a later recovery, and this logging information should be marked `invalid` to avoid a later restart of an erroneous transaction.

The system may also be forced to cancel a transaction if it finds errors, or the user may decide to interrupt a transaction which does not behave correctly. In either case, if the commit point has not been reached, an orderly shutdown presents no serious problem.

Completion After a crash, any transaction which had issued a commit should be completed. This means that the transaction manager needs to have sufficient information to complete the transaction. The transaction may be restarted from its beginning, or completed from its commit point. In order to complete a transaction from the commit point, any data to be written into the database must be kept securely available. In any case some secure storage is needed. We present three alternatives for achieving completion; the first one has two options.

- The *transaction log*, presented in more detail in Sec. 11-3-4, is a separate file which provides secure storage for the data blocks created in phase one and to be written in phase two. Any blocks to be written in phase two are placed on the transaction log before the commit request is granted; these blocks are called *after-images*. Any blocks that do not overwrite prior data in the database are best written directly into the database before commit; the blocks written in phase two will contain references to these blocks so that they too become part of the database during phase two.

When a restart completion is required, the transaction manager copies the after-images into the database. Any blocks already written by the interrupted transaction during its phase two are rewritten by identical blocks, and when the transaction manager's restart action is completed the database is in the desired state. The user should be notified of the successful completion.

An option of the transaction-log approach is to let the transaction manager always write the blocks. At the commit point the transaction turns the resposibility for completion of phase two of the transaction over to the transaction manager. Since the transaction manager already receives the data on the log and has the capability to complete the effect of the transaction, little incremental effort is needed. The transaction itself can terminate earlier and release its resources. The transaction manager will still have to secure the after-images prior to writing into the database. The transaction program and its submitter can be informed of the expected successful completion right after the after-images are safely stored.

- Transactions may maintain their own after-images. Then the transaction manager simply executes a *transaction restart* when completion is required. At the commit point a transaction provides a restart address. When restarted by the transaction manager it carries out its own completion. *Self-completion* is quite feasible if a transaction has some nonvolatile storage available. Some modern semiconductor memories will retain their contents so that such write blocks are retained and restart is feasible without having copied the after-images. This technique requires very careful verification of the transaction's correctness and its capability to

restart. It is rarely done in commercial data-processing but is useful in some scientific environments, where data are acquired rapidly from experiments and cannot be reentered.

- The third alternative for completion is the *reexecution* of the entire transaction. Here the transaction manager obtains the name of the transaction and the input data and places it on the transaction log for safekeeping. By waiting to the commit point the transaction has already performed any checking that could have caused an internal abort, so that no erroneous transactions will be restarted.

If a committed transaction cannot be completed, the transaction must be *undone* to avoid leaving the database in an inconsistent state. The submitter of the transaction should be informed that the transaction has not been carried out. The information required to resubmit a very recent transaction which failed is typically still available at the submitter's site. A transaction manager should always inform the user when a transaction has been successfully completed, since the user at the terminal may have sensed the failure of the system and will be tempted to reenter a transaction submitted prior to a failure.

Undo In order to undo a transaction the prior state of the database must have been preserved. Two techniques are feasible to support undo transactions:

Version creation All new or updated data to be written is placed into previously unused storage. No older data are destroyed. A pointer to the most recent valid data, the *high-water mark*, is updated only after the new data has been successfully written.

Logging and backup All old data are saved on a transaction-log file. When an error requires an undo, the older data are located and the files are restored to a state which eliminates the effect of the transaction. The old blocks saved are referred to as *before-images*. They should be written before the commit point.

We will deal with version creation in Sec. 11-3-3 and with logging and backup in the remainder of this chapter. Problems caused by intermingling of versions or logs from concurrent transactions are addressed in Sec. 13-2-5.

In any case it is important to reduce the probability of failure after a commit is declared. This reduction can be achieved by reducing the number of blocks which have to be written to file after the commit point.

As described when discussing completion, any blocks that can be written without destroying older data are best written before the commit point. After the commitment one or more blocks will be written which make reference to the blocks written prior to the commit point, so that now all blocks become part of the new database state. The transaction will be excellently protected from hardware failures if only one block has to be written in order to link all changed blocks into the active database, since a single block is generally written completely or not at all. This technique already mimics the notions from the versioning concept.

11-3-3 Version Creation

Any time that new data are added or old data are updated we can create a new version of the database. If we save the prior version of a database we obtain a good basis for restoration in case of errors. Periodic copies of database versions are called *backup copies* or *dumps*. They are discussed in Sec. 11-4-3.

If prior data are not overwrittten, previous versions may be kept available without copying the old data to some other device or area, Creation of an entire new version of a database is feasible if the new database consists requires rewriting much of the old data. We see this being done where small databases are created by periodic abstraction of some other database, for instance the list of invalid credit cards of some financial institution. Typically, however, a new version consists of both old and new units of data. Candidate units are areas, files, blocks, and records.

We will consider blocks to be the unit of data. A version of a database is defined as the collection of all the most up-to-date versions of each block relative to some time point. The current version of the database comprises all the most recent versions of the data blocks it covers. New blocks can be simply appended to the file storage. The address of the last block written is the *high-water mark*. A previous version can be recovered by resetting the high-water mark to an earlier state.

The notion of having versions is simple but it is obvious that the storage demands due to keeping data for all past versions can be high. Older data is kept on storage devices with the same cost and performance as current data. Maintenance of efficient access to the set of current blocks requires careful support. The use of versions is becoming feasible and necessary with the appearance of large-capacity storage devices which can write the data only once. Optical disks are the prime candidate for such techniques.

The complexity of dealing with versions relates to our demands for access performance. If a pile-file organization (Sec. 3-1) satisfies our access needs, versioning is simple. A reverse search from the high-water mark can recover any desired block of the database. Such modest requirements are rare; most files require some access organization and each data rewrite will in fact require the update of some index or access information. Issues of access management when using versions have not yet been thoroughly studied.

It appears that three schemes for access management to versions are feasible:

Access and data are mixed The rewritten blocks contain the relevant access information internally. This information should include a new, partial, root index block with paths to recent data and the prior root.

Access indirection The volume of block references is reduced by using indirection. The rewritten block includes only replacement information for indirect referencing, as described in Sec. 8-1-2.

Backup maintenance for access information Only data blocks are versioned; any index or other access information is maintained in core buffers and rewritable storage. When an error occurs, index information is recreated from backup data, including the versioned data.

Support for the last scheme is provided through the same logging mechanisms used for recovery in nonversioned databases, which are introduced in Sec. 11-3-4, but

versioning of the primary data will greatly reduce the logging volume.

Resetting of the database to a prior version does not solve all recovery problems. An effect of a transaction which was carried out after the transaction to be undone will also be lost if the later transaction updated the same block. More information is required if this case is to be handled correctly. Even more problematic is dealing with the effects of concurrent transactions. The activity-logging techniques described in Sec. 11-4 address these issues.

11-3-4 Transaction Execution

Transactions may be update requests with new data or may be queries. Queries do not have a second phase. The execution of update transaction is now accompanied by logging of sufficient information to guarantee abort or completion. The transaction manager, after receiving the commit request, verifies that all data required for completion have been secured and grants the transaction the permit to go ahead. In a distributed database this verification may require communication with multiple or even all nodes.

Saving of the prior database state by logging also makes the undoing of a specific transaction possible. Transaction logging is often incorporated into a larger scheme for database protection, *activity logging*, which will be covered in Sec. 11-4. The specific data elements being logged are described there. Activity logging also considers correction of erroneous output for query transactions.

The data elements logged for a transaction must be identified. The *transaction identifier* must be unique and should be serial if the effect of one transaction upon other transactions is to be evaluated. For this purpose we may assign a sequential identification number to every transaction. A time stamp with a sufficiently small time increment can also provide a sequence number. The date and time of transaction arrival is always useful.

Execution Sequence It is in general important that the sequence in which the transactions were executed matches the sequence in which they may be restored.

If transactions A and B include update operations to the same data, say

```
A:   X = X + 100
B:   X = X * 2
```

reversing their execution sequence during recovery will create new errors.

Many operations found in commercial processing are *compatible*; i.e., their execution order does not matter. Addition and subtraction are compatible, and these dominate in financial and inventory transactions.

Transactions having operations on multiple shared data items still have to be constrained in general and keep the same execution and restart order:

If transactions C and D include apparently *compatible updates* to shared data, say

```
C:   X = Y + 100
D:   Y = X + 50
```

reversing their execution sequence during recovery will also create errors.

Still, in commercial processing many transactions are found which are not affected by changes of their execution sequence. Banking deposits and withdrawals, but

not transfers between accounts, remain insensitive to the execution sequence, as do inventory updates and withdrawals, if they are not accompanied by more complex financial operations. The definition of *sequence-insensitive transactions* becomes a matter of transaction design, and this feature must be indicated to the transaction manager if advantage is to be taken of the insensitivity.

In the absence of information about sequence-insensitivity the transaction execution and reexecution will be controlled rigidly by the sequence numbers and proceed serially.

Execution Control of Distributed Transactions In a distributed system the transaction identifier should include a site number. This will make a locally unique number globally unique. Since not all subtransactions will appear at each node the number of last prior request may be shipped along to permit checking for lost subtransactions between node pairs.

Sequence numbers generated at distinct sites will not have a meaningful global ordering, but a time-stamp from a remote node may not be completely synchronized with the local time. Only in very remote sites should this difference be measurable, and users at such sites will rarely sense the effect. We had best assume in any case that users at distinct sites operate in independent time frames. This notion can be generalized to state that transactions submitted at distinct nodes will always be relatively *sequence-insensitive* to each other.

An adequate global identifier can be obtained by catenating local time and the site number. The execution and recovery sequence will be determined by this identifier. The execution sequence may then differ from the submission sequence by the difference in local and remote clocks used for the time stamps.

The identification numbers which have been created in a distributed system will move with the subtransactions through the network, and excution must occur in the same ascending order at all nodes. To assure a complete global ordering the transaction manager should grant commits only in order of transaction identifier. One node or its communication link may be tardy in submitting commit requests. The transaction manager should check that no logically prior requests from any other node are waiting. This requires sending and waiting for messages from all nodes. A tardy node may still not respond, so that the potential arrival of a logically earlier request is not known and it is not feasible to delay requests from other nodes which have arrived. A majority of responses can be used to let the transaction manager go ahead.

The transaction manager now has to deny delayed commit requests and demand an abort of the transaction. If the transaction manager receives detailed information about all activities required for completion, namely the block identifications of all blocks to be written at each node, the potential for conflict can be analyzed. If no conflict exists, a commit can be granted, even when the remote transaction request was delayed.

Another alternative is that a node wishing to commit requests a sequence number from centralized resource. Even though this scheme identifies all potential requests, it is still possible to have a subsequent failure of the requestor, so that the central transaction manager still needs the ability to skip an assigned sequence number.

11-4 ACTIVITY LOGGING

For proper restoration of a damaged database, it is necessary to reestablish the database files without the effects of an error. In Secs. 11-3-3 and 11-3-4 we have shown two methods, versioning and logging, for obtaining the database state prior to some single transaction failure. A version of the database prior to an error can be obtained by resetting a database with past versions to an earlier state or by the application of all saved *before-images* to the current state. A third alternative to reset the state is the use of a backup file, described in Sec. 11-4-3. We will now also deal with secondary transaction failures and with device failures.

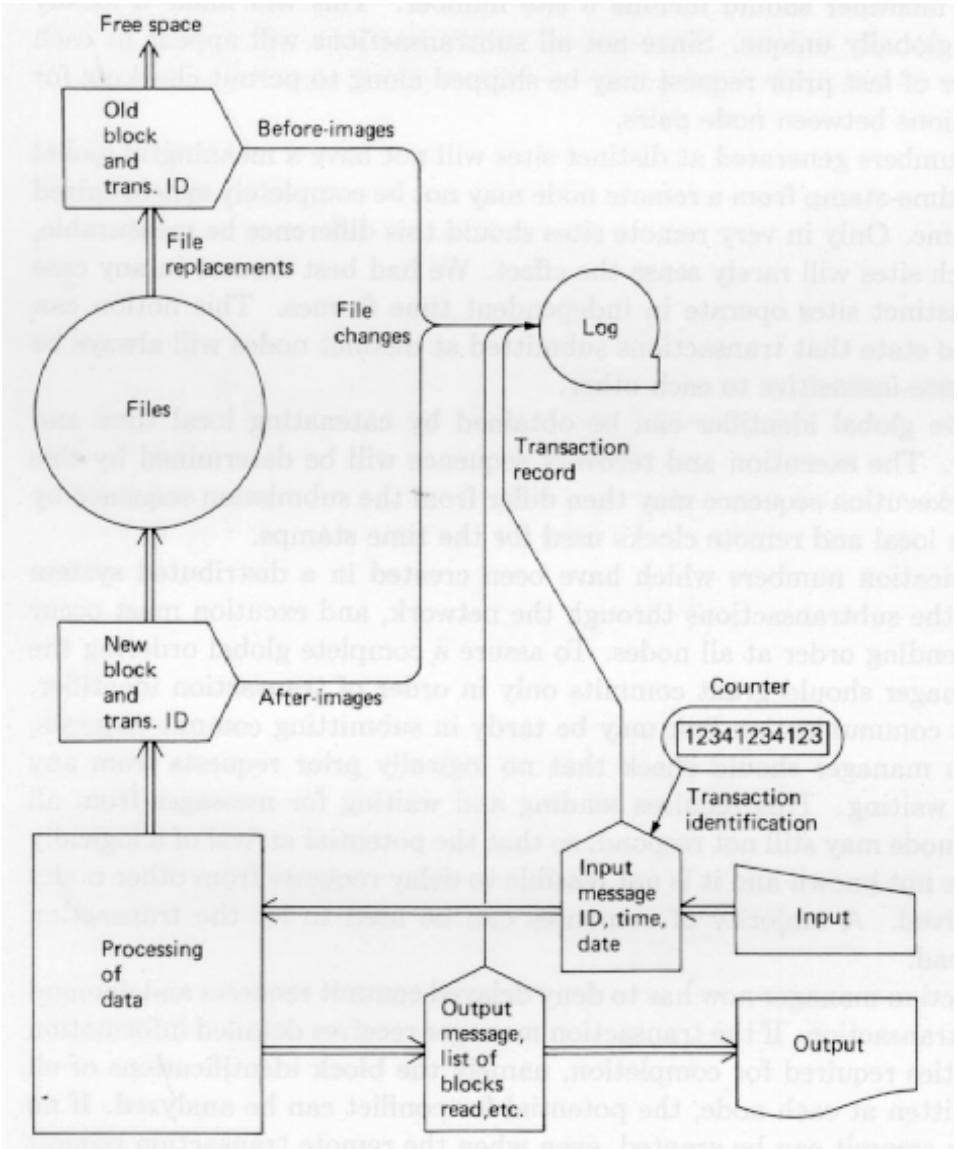

Figure 11-6 Data flow with activity logging.

Restoration of the state to a time before the error occurred will erase the effect of other transactions which caused concurrent or later changes to the database files. Subsequent transactions may, however, have used erroneous data placed into the database from a primary error and can now generate secondary errors.

If a new employee was assigned to the wrong department, any subsequent payroll and budget calculations will be wrong. If these calculations also update the database, the error can spread.

If there was a hardware failure causing actual disk damage, there may not be a current version to apply before-images to.

In order to enable complete restoration, a more extensive log can be kept of all the activities which affect the database. Data are only appended to the log; it cannot be updated. Such a log is often written onto magnetic tape, since on such a tape one can collect economically a sequential record of activities. Tapes also provide mechanically reliable storage, particularly for blocks which are already on the part of the tape which is wound on the receiving reel. Disks are finding increasing use for logs, and can reduce the labor costs. Figure 11-6 illustrates the functions which will be discussed.

11-4-1 Recovery Methods

The purpose of these logs is to allow the recreation of files. Depending on the type of error, database recovery can be performed by

Restoration: combining earlier, correct versions of the file with updates due to correct input received later. Restoration proceeds forward from a past version to the proper current state. Even when versioning is not used throughout, a version or backup copy may be created periodically.

Rollback: undoing effects of errors from the current state until an error-free state is restored. After rollback some restoration may be done.

In either case considerable information must be placed on the transaction log.

11-4-2 Elements to Be Logged

For the protection of individual transactions (Sec. 11-3-1) we required the *transaction identification*, some *after-images* for completion, and if transaction undo is also supported, corresponding *before-images*. For transaction protection the before- and after-images need not be kept longer than required to confirm successful completion of the transaction.

In order to support general restoration or rollback these elements are kept much longer, at least to a point where a complete *database backup*, as discussed in Sec. 11-4-3, is available. Also all after-images have to be logged to assure recovery from storage errors. Additional elements to be logged are the *transaction input*, the *transaction output*, and a *transaction progress thread*. Transaction progress is documented by keeping a list of records or blocks touched by the transaction. All these elements will be labeled with the transaction identifier. A transaction message, all images, and the activity thread due to one transaction are shown in Fig. 11-7.

Transaction input In order to select faulty transactions or find poor input data, the transaction request itself should be logged. The full message text entered, or a code identifying the transaction type and its input, will be copied onto the transaction log.

Replaying of all transactions inputs since the original creation of the database should enable the restoration of any subsequent database state. Any input messages of transaction that caused errors are of course eliminated from the replay. In this manner a recent and proper file status can be established. In practice the cost of such a recovery might be enormous. This technique is useful after a recovery to an earlier state by rollback or may be applied to a backup copy of the database.

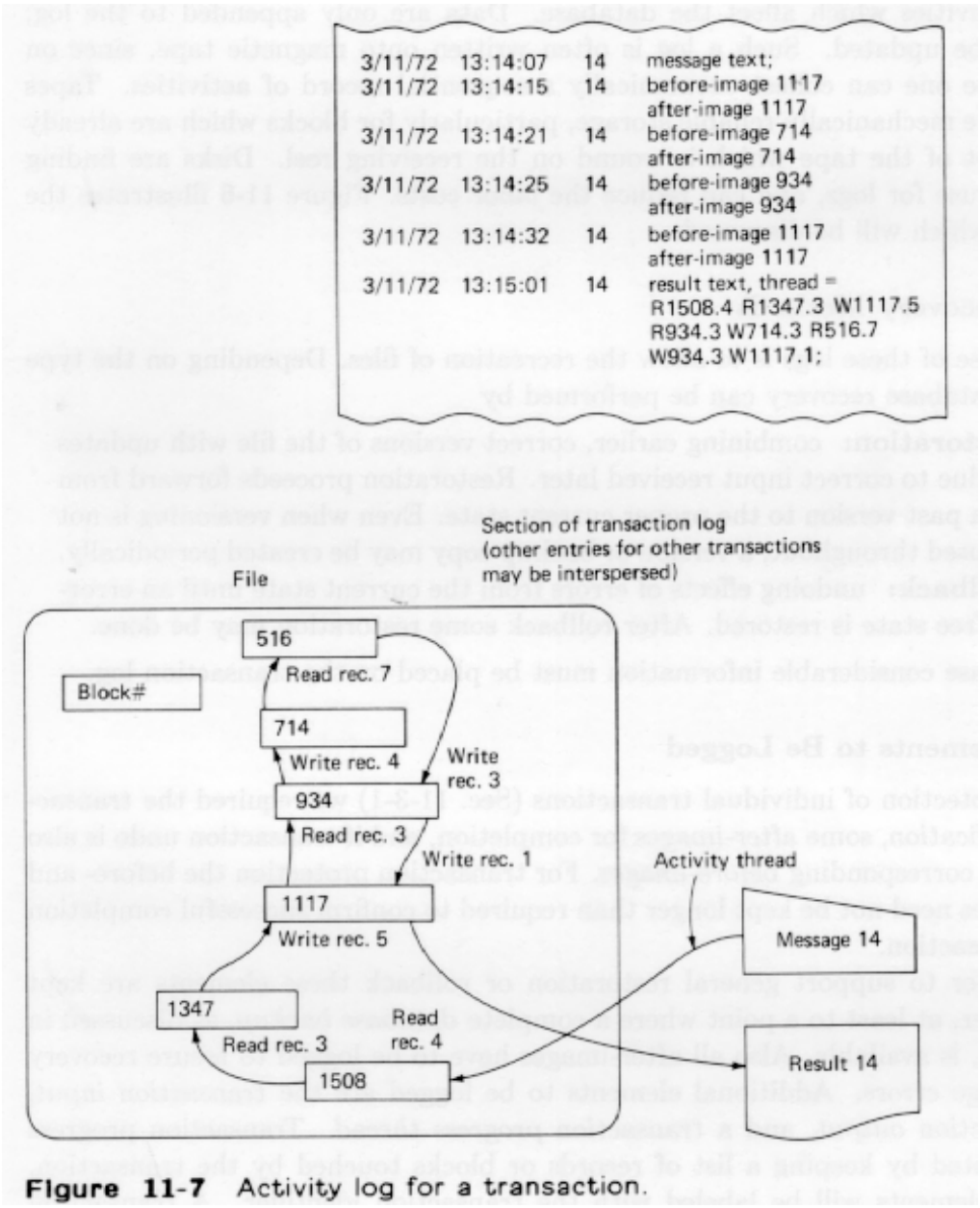

Figure 11-7 Activity log for a transaction.

In some applications transaction input logging may be very difficult. If the messages are long, it may be desirable to copy them on the log by a separate preprocessor before letting them enter the system proper. Communication preprocessors are a feature of many larger systems and can help provide backup. To increase the chance of correct execution the actual transaction processing should be initiated only when the complete message has been received and logged.

In some scientific applications, data are acquired directly from instruments at high rates. Logging of all the data received may exceed available computer resources. In these instances, backup can be provided by collecting a copy of the data near the source, possibly on analog tape recorders. This method, though, will not allow the level of automatic recovery which we can design for the commercial system where messages can be completely copied by the system onto the transaction log.

Transaction output When a transaction is completed and has left the system, another entry on the transaction-log should be made to note the completion of the transaction. This information can be used to prevent the sending of duplicate output generated by correct but reexecuted transactions during a recovery operation. If the full text of the output result is kept, the transaction log can also be used for the correction of output communication failures, which will contribute indirectly to improved maintenance of files.

If, for instance, only the output message text has been lost, the result can be replayed without executing a new transaction request which might cause a duplicate update. A general capability for a user to inquire what happened to a transaction submitted to the system improves the user interface and leads to increased system reliability.

Before-images In order to preserve the prior state of the database for use in rollback, any block acquired with the intention to update and rewrite it must be copied to the log prior to writing of the updated version. The old block written is referred to as the *before-image*. All before-images should be written before the commit point. They are also marked with the identification of their transaction.

Even though only a field or a record may be changed, the entire block may be copied to the transaction log. This done for two reasons: simpler restoration and better protection. If only the record or field to be changed is logged, the restoration requires a merge of new and old records. This merge can be complex if records vary in size, and cause block overflows. Logging only field changes reduces the logging load significantly and is used where performance or storage problems are expected.

These before-images can be used to undo the effects of any erroneous transaction. If there have been no subsequent transactions which have further updated the bad blocks, such a transaction can be cleared, as far as the database is concerned, by replacing all the affected blocks with their before-images.

If subsequent transactions may have used the data, all subsequent before-images may be restored in order to recreate an earlier version of the database; this process is called *rollback*. In order to avoid a complete rollback, the transaction thread presented below must be known.

After-images The use of rollback or of a backup can restore an earlier correct state. To restore a current state, the transaction input can be replayed but reexecution of all earlier transactions will cause intolerable delays. Even if a backup version is available which is perhaps a day or a week old, the processing time required for restoration can seriously affect all database use. We therefore also write on the log the complete result of all transactions as *after-images* and can restore a file from its initial or an intermediate state by copying into the database all after-images. Transactions which accessed bad data will of course not be restored using after-images, since this would restore secondary errors. These must be replayed or the submitters must be advised of their undoing.

After-images are especially useful when files have been destroyed independent of any specific transaction. No transactions have to be reexecuted. The cost of restoration can be reduced by sorting the after-images into the file sequence and omitting all older copies. Since some blocks are rewritten quite frequently the number of blocks to be replaced during restoration can be greatly reduced.

Transaction progress thread The combination of a transaction-log file and before- and after-images provides the capability to recover the files from the current version or from a past version, except where the subsequent use of erroneous stored data by other transactions has further contaminated the file. Results presented to users can also have been wrong if the queries referred to bad data.

In order to determine which transactions were indirectly affected by a file error, it is necessary to keep a record of every block that was read as well as the blocks that were written by every transaction. Since the written blocks have already been copied as after-images and the blocks only read did not change, a simple list of all the affected blocks will suffice. The progress of the transaction is recorded as if a thread were drawn from affected block to affected block. A list defining this activity thread is written out on the transaction log at the same time the transaction-complete message is written, as shown in Fig. 11-7. No other transactions should have had access to the data written between the commit point and completion of the faulty transaction.

The activity thread permits listing all subsequent transactions which read data from blocks generated by the faulty transaction. Any such transaction found is added to a list of affected transactions LA, as shown in Fig. 11-9. The activity threads of affected transactions will also be processed to find yet more affected transactions. The list of affected transactions also indicates which output messages may have been wrong because data from blocks containing errors were used.

Load Due to Logging of File Changes The volume of both the before- and after-images depends on the frequency of updates versus simple read requests to the file. In a typical environment, this ratio is approximately one to four so that the writing of two blocks to tape with every update may still be tolerable. The blocks containing the before-image are already available in memory for processing and record insertion. They can be written from memory before they are changed rather than be copied separately from disk to log file. The after-image is, of course, available in memory for copying to the log when the data file is being written.

Size of Protected Units We have used a block as our basic unit of protection. The smaller a basic unit we choose, the more manageable the problems associated with interference, copying, and restoration become. A record or field, however, is apt to move within blocks and may vary considerably in size. More bookkeeping is required to refine the unit of protection to less than block size. The log may, however, indicate which record of a block was the object of the access to avoid unnecessary restoration. The activity thread should always identify the record, or even the segment or field, so that the set of affected transactions will be kept minimal.

If an error found in a record is due to a system or device failure, it is best to check and restore the entire block, since other records of the block may also have been affected. A software system or transaction fault can destroy adjoining records within a buffer. Section 11-2-7 described a scheme which is especially useful if only field or record images are logged. In Chap. 13 more discussion of the protection unit size or *granularity* will take place.

11-4-3 Backup

The use of replay of transaction input or the restoration via after-images relies in practice on availability of a backup copy of an older version of the database. If versions are used backup copies may be constructed by resetting of the current high-water mark; otherwise backups are created by copying. Backup copies may be periodically generated, and a series of past versions may be kept. Each backup copy will be identified by time and date, and by the last transaction included. A backup copy should be generated while the database is quiescent, since updates during copying may cause the copy to be inconsistent. In a *very large database* there may never be a quiescent period long enough for making a backup copy. The transaction log can help here too.

Consistent Backup File Snapshot A sorted list of after-images, with older versions of duplicates eliminated, will in effect be a copy of a version of the entire database. Such a copy can be created separately from the database processing itself by using collected transaction logs, and will provide a backup for the entire database. This copy can be made consistent to any point in time by eliminating blocks which were changed because of transactions submitted beyond the cutoff point. The transaction-complete entry on the log can be used to determine that all transactions entered previous to the cutoff point had been completed.

Such a condensed list of after-images comprises a snapshot of the database in a form that actually never existed at any point in time. The copy will be *internally consistent*; that is, any transaction will be represented either completely or not at all. In a busy system with a very large database, as discussed in Sec. 9-7-4, there may never be a time where the database is internally consistent. This snapshot is actually more useful for backup than a true and complete instantaneous image during a time when the file was in active use. DL/1, for instance, provides facility to generate *backup* files in this manner.

Program Checkpoints In systems which process large computations, it may be undesirable to repeat the entire computation if a problem is encountered which makes progress impossible. To solve this problem the computation can include *checkpoints* which are initiated by the program. At a checkpoint, the past activity thread is written to the log as well as the entire state of the computation. When a computation has to be restarted from its checkpoint record, the state of the computation is restored and all file devices are reset to their state and position at checkpoint time, so that currency indicators remain valid. The cost of resetting such devices as tapes, and input units is so high that the placing of checkpoints requires great care.

11-4-4 Protection and Debugging

The development of new programs requires testing, and this can place a database at great risk. While it is possible to obtain a copy of the system for purposes of checking out programs, such a copy, especially if it requires many disk units, can be expensive to use. Often the debugging of changes in a large system will disable normal operation of the system. Earlier generations of programmers have tolerated the resulting impositions and often have worked at extremely awkward hours.

When we design a system today with the reliability features discussed above, we can include some features which can make it possible for programmers to rejoin the human race. First we install a separate log file, a *debug-log*, on which transaction information about all debugging transactions is written. Then we disable any output to permanent files or to regular user terminals for transactions being tested. The needed output for checkout will be found on the debug-log. When a transaction of a program in debug status requests a block, a list or hash table is checked to determine if this block was placed onto the current debug log, and if it was, the most recent copy is retrieved from the log by reverse scanning. This debug-log approach is now available to IDS users. While this practice will cause a slower response time for the debugging user, especially if the debug-log is maintained on tape, the system and human cost of such an approach will be immeasurably more reasonable.

It is possible to use the transaction log also for the debug log when testing. Interference with normal operations will occur when a block written in debug mode has to be retrieved. Any restoration has to ignore test transactions.

11-5 A SCENARIO FOR RECOVERY

In the preceding sections we have discussed some of the methods which can be used to collect data to protect the reliability of the database. We now will describe a sequence of actions which could be employed when a system failure occurs. We will assume that we are using a system which provides the logging facilities discussed earlier. Our task is made simpler by the fact that we do not consider the effects of the interaction of simultaneous programs at this point.

The following steps will be discussed:

Error detection
Error-source determination
Locating secondary errors
Application of corrections

11-5-1 Detection and Error Source Determination

The recovery process starts when the existence of an error is detected. A variety of entry points into the recovery process can be distinguished. We will consider system failures, detected by lack of system action or by unrecoverable redundancy checks, and incorrect output, noted by a user.

There may be a *system crash*, either immediately observed by computer operating personnel, who notice that equipment has stopped, or brought to their attention by users whose terminals no longer respond in the expected fashion. A crash has

the advantage that the cause of the error is often obvious and that the time period of incorrect operation has been small.

If an error has been detected through one of the *redundancy*-based mechanisms, the error also has had little chance to propagate. If the error is due to an erroneous data record obtained from the file, a backward search through the transaction can locate the point where the data was put on the file. Such an error has had time to propagate. Manual correction of the erroneous element has to be performed if the error is an input-source error. If the error is due to malfunction of the storage device itself, it is sufficient to locate the after-image which was saved when the block was written and initiate restoration by replacing the bad block. If failures are caused by *access-path* errors, it may be possible to reconstruct the file structure without reference to data portions of the records. An index chain which has been damaged can be reconstructed from the data. If data records themselves are chained, the damage requires reconstructions from the log tapes.

Much more insidious is the failure which is detected only by the appearance of *incorrect output*. There may be a very long time interval between occurrence and detection of the error. Considerable detective work may be required to locate the cause. A search backward through the transaction log can determine all updates and thus find suspicious transactions.

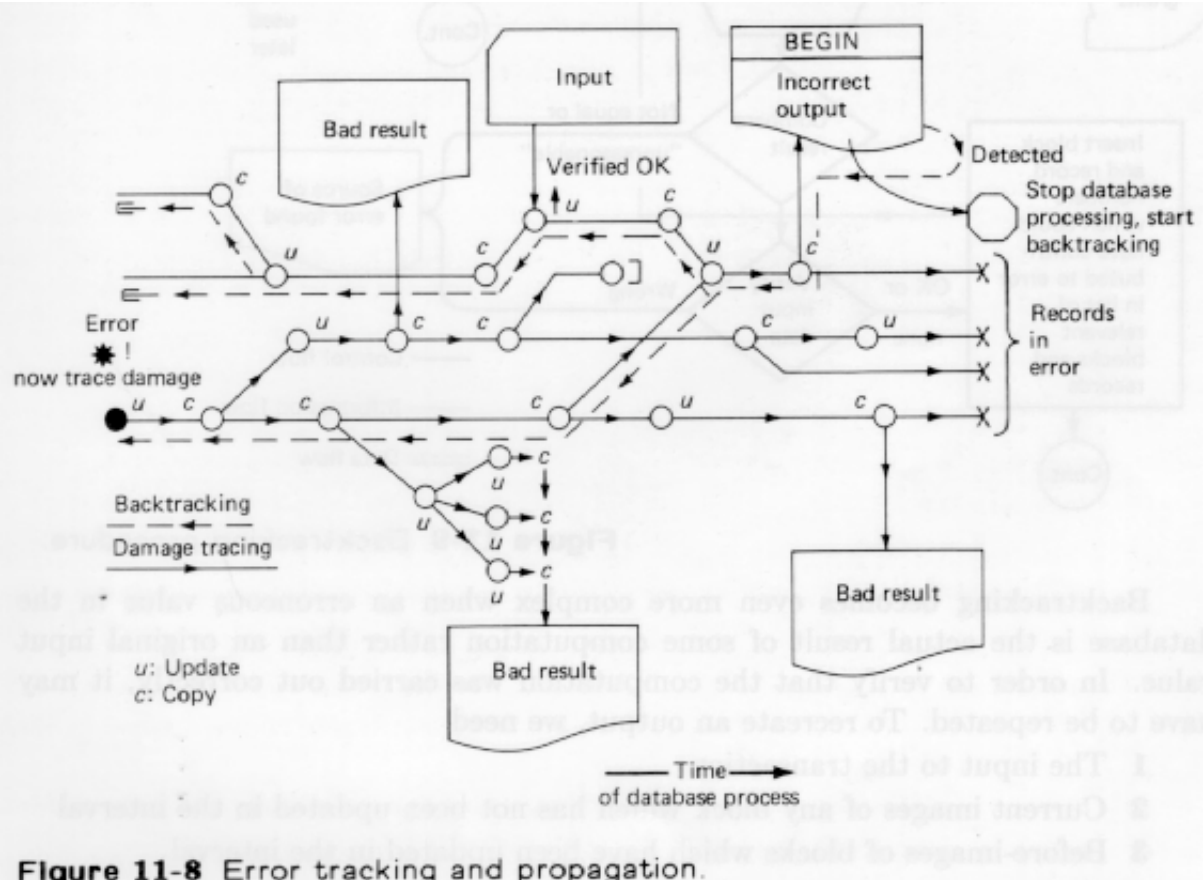

Figure 11-8 Error tracking and propagation.

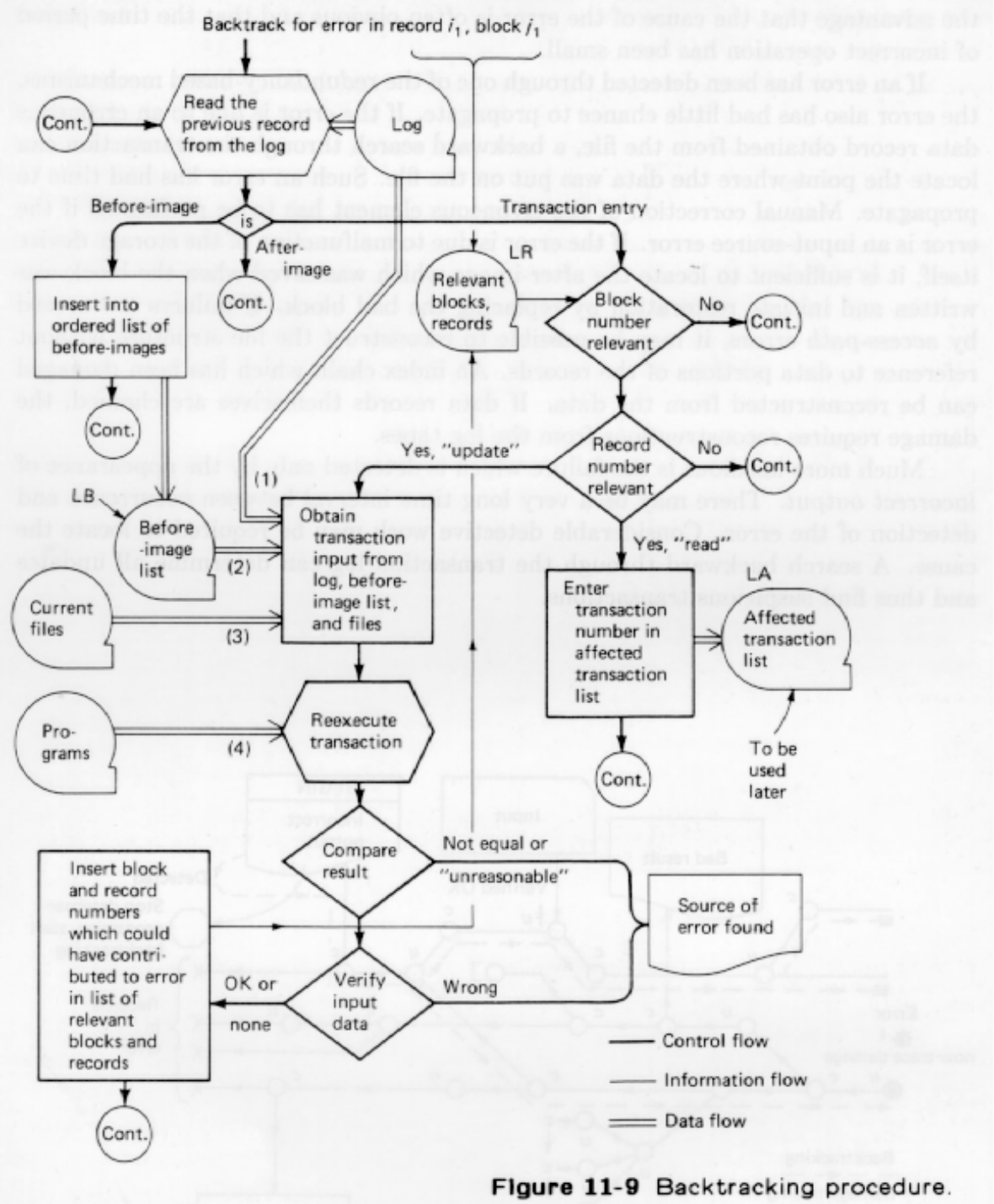

Figure 11-9 Backtracking procedure.

Backtracking becomes even more complex when an erroneous value in the database is the actual result of some computation rather than an original input value. In order to verify that the computation was carried out correctly, it may have to be repeated. To recreate an output, we need

1 The input to the transaction

2 Current images of any block which has not been updated in the interval

3 Before-images of blocks which have been updated in the interval

4 The program used to execute the transaction

The correctness of any input data entering the transaction has to be verified. This verification may also require manual assistance. To determine the conditions for 3 and 4 requires the creation of a list of before-images encountered while the transaction tape is processed in reverse order (LB in Fig. 11-9). Reexecution of a transaction may result in a different output or in a value which is found by inspection to be in error. The determination will probably require human interaction. If no error is found in the update procedure, we will have to continue backtracking.

All data which contributed to the value in error have to be verified. It is rare that we will have available a description of the computation in reversible form, so that we will assume that all records read during the processing of the transaction become candidates for further backtracking (LR).

The automatic backtracking process may proceed in parallel and collect all possible inputs. The lists of relevant records (LR), before-images (LB), and transactions which may have been affected (LA) will grow quickly as we walk backward in time. There will come a point where this process has to be discontinued even though the source of the error has not been found. We may compare the status of the file at that point with the status of a backup copy. If a comparison with the backup file does not indicate an error, an audit or other verification procedure has to be used to determine which data element is wrong and its correct value.

If bad data is found, but determination of the correct value would take an excessive amount of time, it may be easier to delete the input, proceed with the restoration procedure, and apply a correcting transaction later. The ability to apply corrections is basic to most systems, since some human errors will always occur.

Determination of Extent of Damage In order to decide which corrective action is best, the extent of the damage has to be determined. This effort is, of course, closely related to finding the time and the cause of the error. After a *crash*, or when processing has been interrupted because of an error signal, we need to determine both which areas of the data file are suspect and which transaction did not complete. Blocks recently written may be checked for redundancy errors and may be compared with their after-images. If any errors are found, the related transactions are marked incomplete.

The backward search through the log can provide a list of any transactions started and not completed. If they performed any file updates, before-images can restore the files to their previous state.

When errors extend back into the file, a subsequent forward pass through the log can be used to collect all possibly affected transactions. The table (LA) collected during the backtracking procedure illustrated provides the initial entries for this search.

11-5-2 Locating Secondary Errors

When an error which caused an improper change to a file has been detected, a scan through the activity lists will find the transactions which used the bad block. The affected transactions can be then reentered automatically and correct results can be produced. If blocks were updated by transactions which read bad blocks prior to the write, further restoration of the file is required. The identification of all blocks and

all messages affected can be produced by an iterative search of the transaction log. Comparison of the block or the result which was written with the correct version, to ensure that there was indeed an error, can avoid needless duplication and prune the search tree for secondary errors.

Since a transaction which uses internal data, in general, updates only a few records, the list of secondarily affected transactions should be small, especially when an error is detected early. If the damage is extensive, rollback restoration by replacing blocks using before-images may not be practical, and a forward restoration starting from a previous file snapshot which uses after-images may be a better procedure. The decision of rollback versus forward restoration may be automated, although one should hope that the decision is not frequently required.

The activity thread created for file-restoration purposes not only provides the means to recreate the file, to notify users, and to correct erroneous messages but also can aid in the maintenance of file integrity, as will be discussed in Sec. 13-2.

11-5-3 Application of Corrections

If the extent of damage is limited, a rollback process can be used. The damaged portions of the file are restored by first applying any before-images to the blocks in error and then replaying the incomplete transactions. The output from these transactions is suppressed, if possible, to avoid duplicating results that have previously been sent to the users. If this capability to select output does not exist, this output is marked as being due to an error-correction procedure and the user will have to determine its relevancy.

If the damage is great, one may have to start from a backup copy of the file and apply all after-images to it in chronological sequence, except those associated with incomplete transactions. Incomplete transactions are again replayed.

An overview of the process is sketched in Fig. 11-10. A program for the recovery process should be *idempotent*, that is, be able to restart itself at any stage, so that failures during recovery will not become disasters. During recovery the systems tend not be stable, and failure rates higher than during routine operations. If the restoration is major, it may be wise to copy the portion of the database to which corrections will be applied since restoration procedures, being rarely used, have a substantial chance of failure.

11-5-4 Recovery in Distributed Systems

We considered up to now program and input errors, as well as problems due to processor failures. In distributed systems we also have to consider communication failures. These change several assumptions made earlier.

Systemwide operation A part of the distributed system may have failed. Transactions can still be submitted at some nodes, and some may be able to complete while others cannot.

Failure knowledge A block may be transmitted over a communication line for writing or a remote file. The sending transaction may not know for a long time if the block has been correctly and safely stored.

The increased parallel activity from multiple nodes and from the longer times needed

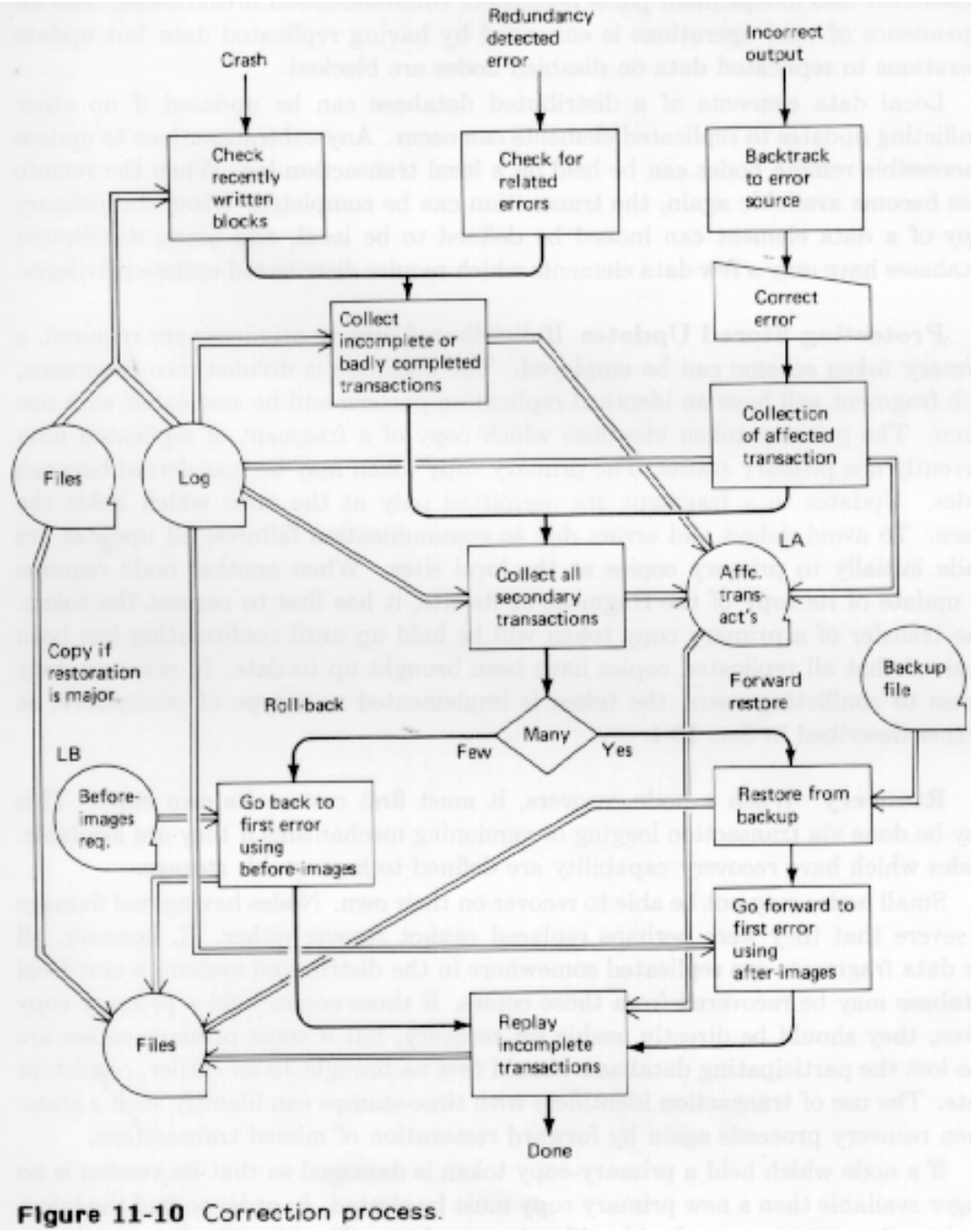

Figure 11-10 Correction process.

to complete transactions increases failure risks in distributed databases. A further consideration is the greatly increased use of replicated data in distributed databases (Sec. 7-5-1).

Operating an Incomplete Distributed Database A motivation for distribution is autonomy of operation in the distinct nodes. Having replicated data permits some continued operation when a node has failed or if the database has been partitioned into independent parts because of communication breakdowns. The independence

of read operations is enhanced by having replicated data but update operations to replicated data on disabled nodes are blocked.

Local data elements of a distributed database can be updated if no other conflicting updates to replicated elements can occur. Any subtransactions to update inaccessible remote nodes can be held on a local transaction log. When the remote sites become available again, the transaction can be completed. Often the *primary copy* of a data element can indeed be defined to be local, and many distributed databases have only a few data elements which require distributed update privileges.

Protecting Stored Updates If distributed update privileges are required, a *primary token scheme* can be employed. The database is divided into fragments, each fragment will have an identical replication pattern and be associated with one token. The primary token identifies which copy of a fragment of replicated data currently has primary status. The primary copy token may be transferred between nodes. Updates to a fragment are permitted only at the node which holds the token. To avoid delays and errors due to communication failures; all updates are made initially to primary copies at the local sites. When another node requires an update of its copy of the fragment at its site, it has first to request the token. The transfer of a primary copy token will be held up until confirmation has been received that all replicated copies have been brought up to date. In order to deny access to conflicting users, the token is implemented as a type of *semaphore*, as further described in Sec. 13-1.

Recovery When a node recovers, it must first restore its own state. This may be done via transaction logging or versioning mechanisms if they are available. Nodes which have recovery capability are defined to have *stable storage*.

Small nodes may not be able to recover on their own. Nodes having had damage so severe that they were perhaps replaced cannot recover either. If, however, all the data fragments are replicated somewhere in the distributed system, a new local database may be recovered from those copies. If those copies hold a *primary copy token*, they should be directly usable for recovery, but if some primary copies are also lost the participating databases should first be brought to an earlier, consistent state. The use of transaction identifiers with time-stamps can identify such a state. Then recovery proceeds again by forward restoration of missed transactions.

If a node which held a primary-copy token is damaged so that its version is no longer available then a new primary copy must be elected. In order to find the latest version of a copy we use the identification numbers. The identification number is included with every subtransaction and also with each data unit updated. The copy with the highest identification number is the candidate for becoming the new primary copy.

To assure that the most recent primary copy can indeed survive, the original transaction cannot commit until there is a high confidence of permanency. Survival may be taken for granted when either a node with stable storage has indicated its readiness to commit or when sufficiently many simple nodes have indicated their readiness to commit a copy to their data storage. Since even stable storage systems are subject to some risk of total disaster the commitment decision may be based on a count $C = 3 \cdot$`#(stable nodes)+#(storage-only nodes)` with perhaps $C > 4$.

Once the damaged node is restored, the transactions and subtransactions which could not be fully completed because of network damage must now be restarted to achieve consistency throughout the database.

Although restoration of a distributed database appears complex, a careful allocation of update privileges can greatly reduce the degree of interdependence of the nodes, so that only few, and perhaps no update conflicts occur. We will see in Sec. 12-3 that the use of database schemas can provide a precise specification of update privileges in terms of update types and update objects. It is more costly in terms of conflict resolution to control simultaneous access to entire files. Problems due to simultaneous conflicting updates are considered in Sec. 13-2.

11-5-5 Problems

Unfortunately, it is true that errors rarely occur singly. Especially serious is a situation where the log or the backup itself cannot be read properly. Most vulnerable are the most recent records. In a crash some of these may be lost in buffers which have not yet been written. In other cases there may be undetected failures in the unit, typically a tape drive, used to write the backup and log files. The latter condition can be largely prevented by forcing regular reading of the log file; such use will provide feedback on the quality of the written file. A log file can also be used by the database administrator to collect data for detailed usage and accounting information, for system-optimization data, and for evaluation of developing equipment needs.

It is especially easy to lose recent transaction input when a failure occurs. When intelligent terminals or a communication network with buffering capability is used, provisions can be made to request retransmission of recent transaction initiating messages as of a given point in time.

The major cause of the tendency for multiple errors to occur is inadequate attention to single errors. Weak operating practices not only encourage errors but also will often leave the systems unprotected because of lack of adequate backup procedures. Ad hoc fixes, poor definition of responsibilities, and assignment of maintenance duties to staff which works also on urgent development tasks have made many database operations a sham. Lack of documentation is often exacerbated by personnel loss. Where efforts were made to avoid these problems, however, database systems have worked for many years without any failure affecting end users. In an operation which wishes to provide a high-level service, a very careful analysis is worthwhile to obtain confidence that reliability is adequate in specific areas, and that all areas have been covered.

Even when full logging capability cannot be provided, the use of adequate detection techniques can limit the propagation of errors to an extent that is manageable. When transaction inputs cannot be retained, it may be possible to clarify the system capabilitites with the users so that they can retain records of their transactions for some period. Users cannot be expected, though, to keep more than a week's worth of backup available.

A usable description of expected practical reliability can help optimize the efforts expended by users and system staff to provide a productive environment.

Neither the attitude of *"our computers are perfect"* nor the doomsday attitude *"beware, all ye who enter data here"* is helpful when trying to get work done.

BACKGROUND AND REFERENCES

Reliability was the greatest concern of the users of early computers; problems were always run several times to verify correctness. After the programming systems stabilized, users began to trust them with large quantities of data, and demands of reliability became more stringent. Protection and integrity became a major concern with the development of multiuser systems, where one failure can impact dozens of users simultaneously, early contributions to understanding of the issues are by Ware[67] and Wilkes[72]. The occasional failures continue to have a major impact on database users. The relationship of load and failure rates is convincingly demonstrated by Iyer[82].

Physical security is stressed in guidelines for federal installations (NBS[74]) and is an important aspect of an AFIPS sponsored report and checklist (Browne[79]). A major study was sponsored by IBM(G320-1370 to 1376)[74] and contains many suggestions and experiences, as well as a very detailed bibliography, but does not provide a consistent model. Fernandez[81] provides a text.

The field of error-correcting codes is very well developed. TMR was proposed already by Von Neuman; the low cost of processors makes this solution increasingly practical, as seen in the space shuttle (Sklaroff[76]). A basic textbook is Petersen[72]; for recent references a collection of papers on coding theory, Berlekamp[74], may be consulted. Brown[70] presents the method used for tape-error control and Tang[69] covers decimal input checking schemes. Reliability measures for data input are stressed by Stover[73].

Techniques to keep files secure are presented by Barron[67] for ATLAS, Fraser[69], Weissman[69] for ADEPT, a military prototype, and Frey[71].

Yourdon[72] and Martin[73] present security techniques. Karsner in Kerr[75] presents experience at the Netherlands PTT. Often reliability measures are discussed with the objective of improving the protection of privacy (Conway[72] and others in Chap. 12). The frequency with which back-up files or checkpoints should be taken has been considered by Vold[73].

Many current DBMSs provide rollback capability. Appendix B lists several. The extent of reliability coverage can be hard to ascertain from manuals, but practical experience has been quite favorable. Verhofstad[78] provides a thorough analysis. Recovery-procedure concepts are presented by Lockeman[68], Oppenheimer[68], and Droulette[71] for IBM OS VS, Fichten[72] for WEYCOS, Smith[72], and Thomas[77] for pointers in EDMS.

The model used here was influenced by work of Iizuka[75], as applied to FORIMS. Severance[76] uses differential files to maintain backup versions incrementally. A technique for creating backups suitable for very large, active databases is presented by Rosenkrantz in Lowenthal[78]. Recovery using cooperation among processes is defined by Kim[82]. Recovery in SYSTEM R is described by Gray[81]. The block-level recovery scheme, using checkpoints in multiple timeframes, and its performance, was detailed by Lorie[77]. Audittrails are discussed by Bjork[75]; Sayani in Rustin[74], Menasce[79], and Bhargava[81L] have evaluation models for recovery actions.

Access to prior database versions is developed by Adiba[80].

The two-phase model of transaction processing is due to Eswaran[76]. It is evaluated by Menasce in Aho[82]. Undo of transactions is supported by an algorithm of Reuter[80].

Reliability in distributed databases is the focus of a series of conferences sponsored by the IEEE (Bhargava[81], Wiederhold[82]). Fischer[82] sets proper checkpoints in multiple sites. The primary copy token scheme is due to Minoura[81]. Reliability in various distributed system implementations is considered by Dadam[80], Hammer[80R], Borr[81], and Andler[82].

Journals Articles on database issues are found in nearly every scientific publication. In Chap. 8 we listed some specific database publications. General computer science journals which frequently carry papers on database topics include the *Communications* (*CACM*, since 1961) and *Journal* (*JACM*, since 1968) of the ACM. Survey articles appear in the ACM *Computing Surveys* (since 1969) and in *Computer* (since 1970) of the IEEE Computer Society. Some database papers also appear in the IEEE *Proceedings* and the *Transactions on Software Engineering* (since 1975).

The British Computer Society's *Computer Journal* (since 1962) and *Software Practice and Experience* (since 1973) often have pragmatic papers. The Australian Computer Society has a *Journal* which carries also database articles. *BIT (Nordisk Behandlings Informations Tidskrift)*, published since 1976 by the Regnecentralen (Copenhagen Denmark) and *Acta Informatica* (since 1971) often have relevant algorithmic papers. Papers on computational topics are found in SIAM *Journal on Computing* and also in *Information Processing Letters* (since 1971).

The ACM-SIGIR produces a newsletter for information retrieval. Oriented towards commercial data-processing is the *ACM-SIGDB Data Base*, produced since 1969 by the SIG on Business Data Processing and the *EDP Analyzer*, produced since 1964 by Canning Publications, Vista CA.

Many applications areas have journals which emphasize issues of data management. Management oriented papers are published in the Journal of the Society for Management Information Systems (Chicago IL). In the medical area we find *Methods of Information in Medicine* (Schattauer Verlag, Stuttgart FRG) and the *Journal of Medical Systems* (Plenum Publishing). The library field uses the *Annual Review of Information Science and Technology*, published by Knowledge Industry Publications, White Plains NY, under sponsorship of the American Society for Information Science. Butterworth, London, has begun publication in 1982 of *Information Technology: Research and Development*.

EXERCISES

1 Investigate the type of protection provided at your computer facility. Is it adequate for current use? Is it adequate for a permanent database? Can users augment the services to provide more protection for their work? How would the costs compare for user-managed protection versus system-managed protection?

2 Devise a check-digit method suitable for

a Dates encoded as month/day/year

b Sequentially assigned identification numbers

c Dollar amounts

3 Design a method to provide TMR for some vital part of a database system. Use flowcharts or a procedural-language description to document the design.

4 Simulate the TMR design of Exercise 3 using randomly occurring errors at several rates.

5 Obtain a description of the recovery procedures from a database or file-system supplier, describe the approach in simple and general terms, and evaluate its effectiveness.

6 Mark in the flowchart made in Exercise 21 of Chapter 4, (refering to Figure 3-40) a dozen possible error conditions. Make a table with entries giving suitable diagnostic messages for each error condition and a code indicating if the error is likely to be due the user (U), the application designer (A), the operating system (S), the file method (F), or the hardware (H).