Feature analysis of selected database recovery techniques*

by BHARAT BHARGAVA and LESZEK LILIEN
University of Pittsburgh
Pittsburgh, Pennsylvania

ABSTRACT

Database recovery techniques in a real-time environment for so called single-division databases are investigated. A classification of database recovery goals and a classification of database system crashes is presented. It is shown that the (best) recovery goal is a function of a crash category against which the system is to be protected. In particular, for the broadest category of hidden hard crashes an actual past state is an attainable recovery goal. It is described how to reach this goal using a generic recovery technique, based on an idea of a database recovery block. The specific recovery techniques implementing the generic technique are described. Then the representation of each specific recovery technique in terms of atomic "primitives" is demonstrated. The claim is made that this "divide-and-conquer" approach can facilitate the analysis of the database recovery techniques.

INTRODUCTION

Database technology is one of the most rapidly growing areas of computer science.[10] The technology makes it possible to reduce data redundancy, as compared to independent file systems, simultaneously improving data availability. But it also introduces the potential for disaster; the database is now more vulnerable to destruction through hardware and software malfunction. The loss of "quality" in a database, especially its total destruction, may be considered a threat to the organization owning the database, because data is one of its most vulnerable assets. The problems can be further aggravated if a database system is to function in a real-time environment. This case is investigated in this paper.

To avoid confusion let us indicate the meaning of the words fault, error and crash (failure) as used here.[4] A fault is a malfunction in a hardware, software, or human component of the system that may introduce or allow to be introduced errors. These are items of data or pieces of program incorrectly stored or transmitted within the system or lost altogether. In due course, an error may cause a crash, which is cessation of normal, timely operation by all or part of the system, or delivery to the outside world of incorrect data. We interpret a detection of an error at time t as a crash at time t; the moment an error is detected, the system must take some special actions and its normal operation is disrupted. But it is also possible that some crashes will become manifest directly and not through detection of errors that cause them. For example, in the case of a major hardware breakdown the fault and the crash are simultaneous.

Clearly it is impossible to avoid hardware or software crashes in any computing system. Thus the only way to protect a database is through the use of recovery techniques that allow one to restore the correct database state in the case of partial or total database destruction.

The steps in the ideal recovery process could be as follows:[4]

- the fact that the system has crashed is recognized, either through error detection or directly,
- the type of crash is determined,
- the faults in the system which caused the crash are identified,
- the extent of the damage is determined, in the database, programs, system files and elsewhere,
- a method of recovery is selected,
- faulty programs and hardware units are repaired,
- the database is repaired or reloaded, as appropriate,
- restart programs are run which reset the state of the system, undo and reprocess any incorrectly applied transactions, and re-open contact with the users, and
- normal processing is resumed.

Many of the above operations are so complex that we do not know how to implement them. After a crash, hard detective work must be done to diagnose the original fault. If it is a hardware problem, some help might be had from diagnostic software or test equipment. Locating software faults is usually more difficult. Many of the faults which occur in real-time systems are transient, depending on particular combination of events and input data. They may be extremely difficult to reproduce, and if they are ever traced at all, it is usually as a result of ingenuity or guesswork by the maintenance programmer.[4] It is not a surprise that such informal diagnostic methods, let alone correction methods, are far from on-line implementation. Yet there exist some approximate solutions to the problems of on-line diagnosis and correction of faults.

*This work was partially supported by the grant GTRS 5680-C-00026 of the U.S. Department of Transportation.

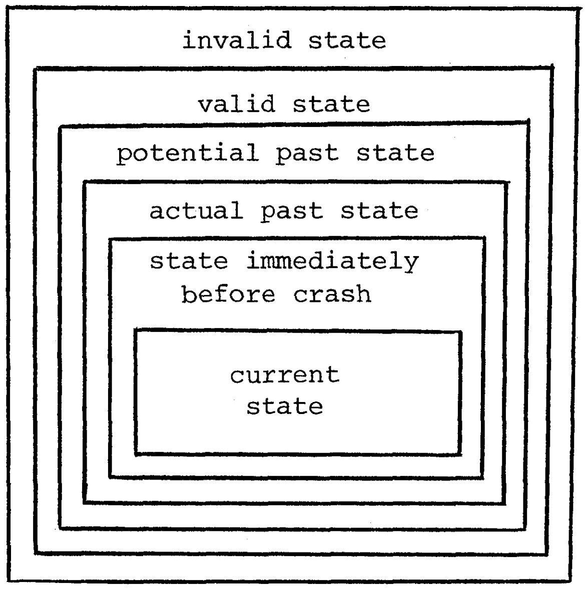


Figure 1—The hierarchy of database recovery goals

In the following considerations the notion of a division is used. We define a division as a logical subset of a database such that integrity assertions of different division are mutually disjoint, and the sum of all divisions constitutes the database. In general, database systems can have more than one division. If an error is detected in one division, it is possible to carry out the recovery process for that division alone. While a given division is under recovery, incoming transactions for other divisions may continue to be processed. This approach increases system availability but makes the recovery procedures more complicated.

We are going to refer to a method for increasing reliability of software system components, which is proposed by Randall.[8] Its basic idea is that all procedures are encapsulated in so-called recovery blocks. Each recovery block comprises a predicate called "acceptance test" (AT) and a collection of alternative procedures for accomplishing the same task. On entry to a recovery block, the primary alternative is tried. If it succeeds, i.e. passes the AT, a normal block exit follows. If it fails, all variables are restored to their values on entry to the recovery block, then the second alternative is tried, etc. (In general, an acceptance test has a limited ability to detect errors, so it is possible that erroneous results pass the test.)

After classifying database recovery goals and classifying database crash categories in the next two sections, we show that the best attainable database recovery goal is a function of a crash category. Finally, we present some selected database recovery techniques and analyze their features.

DATABASE RECOVERY GOALS

Each database recovery technique can be viewed as containing three phases:

1. backing up to a past state;[11] this phase names goals of database recovery, as presented below,
2. restoration of the immediate before-crash state (even if this state is not known explicitly),
3. reexecution of after-crash database operations.

The hierarchy of the database recovery goals, somewhat different from the one presented by J.S.M. Verhofstad,[11] includes (see Figure 1):

1. The current (after-crash) correct database state (this can be a recovery goal only if the database is completely protected from crash effects),
2. the correct database state as it was immediately before crash (what "immediately" means is defined by a single update operation),
3. the actual past database state, i.e. a snapshot of the correct database as it was some time ago ("some time ago" will be defined more clearly by the notion of Database Recovery Block),
4. the potential past database state, i.e., a correct state that is a combination of actual past states of database divisions (these states of database divisions could never exist at the same time, but each of them did exist at some time in the past),
5. the valid database state, in which only a proper subset of database divisions is in a correct state, and
6. the invalid database state, in which all database divisions are incorrect (for a single-division database this goal is equivalent to 5).

The goals higher in this hierarchy, i.e., those with smaller indices, are more difficult to attain than those below. This interpretation underlies Figure 1.

At first glance it seems that goals 1 through 4 are defined in the dimension of time while goals 5 and 6—in the dimension of the database correctness. Our claim that all goals 1 through 6 are really related in one dimension—that of the database correctness—is based on the following approach: more recent correct database state is "more correct" than any previous correct database state.

DATABASE SYSTEM CRASH CATEGORIES

For our purposes we distinguish the following crash categories:

1. soft crashes, i.e. crashes which do not damage the database contents,[3,5]
2. hard crashes, i.e. crashes damaging database contents,[3,5] which may be divided into
   a. overt hard crashes, i.e. crashes that are caused by instantaneously detectable errors or faults, and instantaneously detected after they occur, and
   b. hidden hard crashes, i.e. crashes that are caused by errors detected only some time after these errors occurred. For example, if an erroneous data item is written into a database, this crash can remain hidden for a long time before it is recognized through the detection of the original error or its consequences.

In real-life situations, very few crashes are hard.[5] However hard crash recovery is very time consuming once it happens. Clearly, hidden crashes are the most dangerous; as long as underlying errors are not detected, their effects continue to contaminate a database.

DATABASE RECOVERY GOAL AS A FUNCTION OF CRASH CATEGORY

Protecting against all possible types of crashes is in most cases impractical.[3] This implies the importance of a function

$$\text{recovery goal} = f(\text{crash category})$$

We understand this shorthand notation in the following way. Given a crash category against which we want to protect the database, we aim to achieve the best attainable goal for this crash category. For example, for soft crashes goal 1, which is the best, can obviously be reached. For hidden hard crashes only goal 3 can be reached; more precisely, we will show in this report how to attain the goal 3 and we do not know how to attain better goals, namely 1 or 2, for this crash category. In this sense a crash category implies a recovery goal.

Recovery goal 1, the current (after crash) database state, is attainable for soft crashes. The generic "technique" is just null.

Recovery goal 2, the immediate before-crash database state, is attainable only for overt hard crashes. For crashes of this category we record the database state (e.g., just the old item value) before each update. If the crash happens during the update, we simply restore the old item value. This achieves the required database recovery goal, as a crash is discovered instantaneously. Specialized techniques for recovery goal 2 are not investigated here.

Recovery goal 3, the actual past database state, is attainable for the much broader category of hidden hard crashes. The generic recovery technique and its most prominent implementation approaches are discussed in the next section.

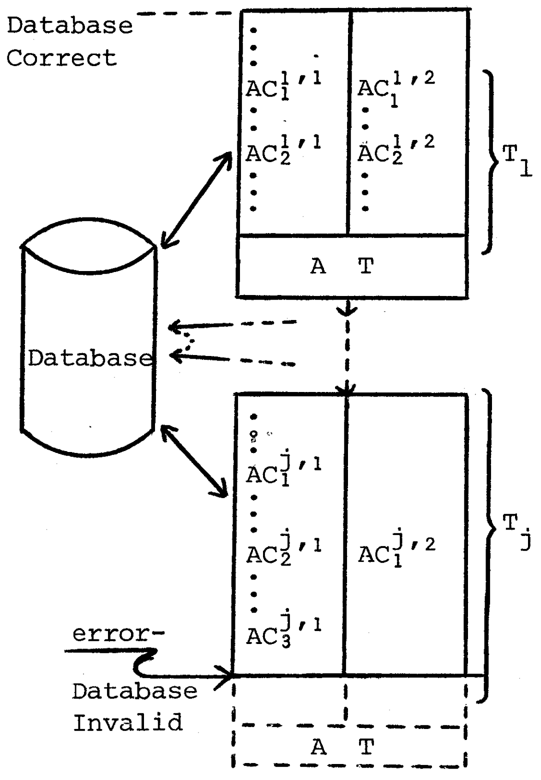
Recovery goal 4, the potential past database state, is not considered here. It seems to be of value in a multi-division database, whereas we assume below only a single-division database.

Recovery goal 5, valid database state, and especially goal 6, invalid database state, are of no practical value—they leave the database seriously damaged and completely destroyed, respectively. Because the proposed mechanism allows us to maintain at least potential past database state, the valid and invalid database states should be seen merely as a closure for the theoretical classification of the database recovery goal hierarchy.

TECHNIQUES FOR RECOVERY FROM HIDDEN HARD CRASHES

Generic Technique for Recovery from Hidden Hard Crashes

For the hidden hard crashes, the database is being contaminated, from the moment an underlying error occurs to the moment it is detected, through an uncontrolled propagation of incorrect database entries.



T_i	-	i-th transaction,
AT	-	acceptance test of a transaction,
$AC_k^{i,j}$	-	k-th acceptance check within the j-th alternative of T_i

Figure 2—Database access by a sequence of transactions—"unsafe" system approach

One means of error detection is the use of acceptance checks (AC). Each transaction can include a number of acceptance checks. Acceptance checks are predicates on values of database items and values of variables of the transaction. It is important to discriminate between acceptance checks (AC) and the acceptance tests (AT) of a recovery block; the former can be placed anywhere in the body of the transaction, the latter are placed only at the exit from the recovery block implementing this transaction. Acceptance checks are specialized error-detection mechanisms (looking for only some types of errors) that can be used to decrease the time interval between the error occurrence and the error detection. Acceptance tests should be able to detect all kinds of errors, but they allow errors to remain undetected until the very end of the currently executed transaction alternative. In general, acceptance checks guarding against more specific types of errors have more diagnostic power. This fact is really not exploited in our preliminary mode.

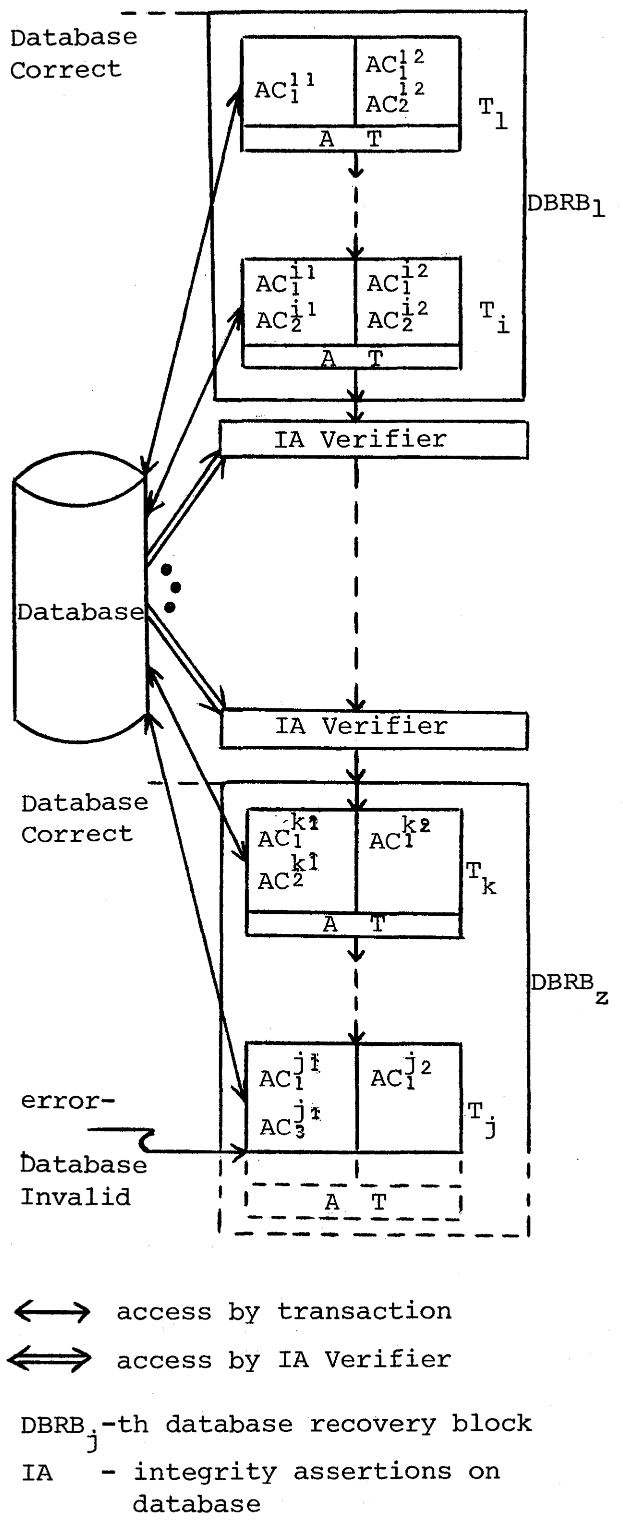


Figure 3—Database access by a sequence of transactions—"safe" system approach

Let us present two scenarios of database system operation. We start with a pessimistic scenario for the "unsafe" system operation (see Figure 2). By definition, the database is initially in a correct state. The processing starts with the transaction (process) T_1. Let the first alternate of T_1 include two acceptance checks ($AC_1^{1,1}$ and $AC_2^{1,1}$) on the database, which give positive validation of its contents. At some moment the transaction T_j starts. The first two acceptance checks positively validate the database. But the third one ($AC_3^{j,1}$) finds out that the database is severely damaged.

The diagnosis is generally impossible. Each acceptance check verifies database integrity only partially. So the fact that $AC_2^{j,1}$ answered positively gives us no help—maybe $AC_2^{j,1}$ did not at all check the integrity constraints that $AC_3^{j,1}$ did. We know only that a hidden crash has happened, caused by an error that occurred after the moment T_1 has started and before the moment $AC_3^{j,1}$ of T_1 has detected the error. It is important to point out that even if there are backup copies of the database in the system, in the case of a hidden hard-crash we have no guarantee that any of these copies is correct. Therefore we cannot use them for recovery.

The only recovery possible in this case is the total system abortion, followed by the restart from the point where T_1 has started initially. The chances that the crash will not happen again are based on the chance that the underlying error is transient or that the transactions are implemented as recovery blocks.

Let us now present a pessimistic scenario of the "safe" system operation. We propose a mechanism confining the database contamination so that a faster recovery is possible than for the "unsafe" system operation (see Figure 3). The assumption of crash transience or implementation of transactions according to a recovery block scheme is still essential here. (Note that not only acceptance checks but also acceptance tests are generally not completely effective; the transaction results could pass their acceptance test and still be a source of errors.)

The system components T and AC are as described above, but new system components have been added. A Database Recovery Block (DBRB), defined dynamically by time interval, encompasses a number of transactions. A DBRB is created in such a way that we are assured of a correct database state at the entry to this block. Before entering a DBRB other than the first one, where integrity assertions are true anyway, the integrity of the database is verified by means of Integrity Assertion (IA) Verifier. (The efficient organization of the IA verification is a problem in itself. What one needs is minimization of the number of database accesses for purposes of the verification. We plan to investigate this subject later.)

After the positive IA verification a database logical snapshot is made and the new, zth DBRB is initiated. (In the case of a negative IA verification, the previous snapshot is restored and processing restarts from that point.) Suppose that $AC_3^{j,1}$ in T_j, which in turn belongs to $DBRB_z$, discovers that the database is invalid. Diagnosis is immediate; a hidden crash has happened, caused by errors that ocurred during the execution of the current DBRB, i.e. $DBRB_z$. The following steps are taken to resume processing:

1. Transactions are not allowed to query the database. All incoming transactions are queued.
2. The transactions, which are implemented as recovery blocks, are reconfigured; some permutation of their alternatives is scheduled for execution. This permutation

is different from that of the ones that are marked so far as trouble-makers and are placed on a suspicion list. (The suspicion list can be used for an off-line diagnosis and repair of transactions. The repaired transactions are removed from the list.)

3. The most recent database snapshot is restored.
4. All transactions that
   a. were active at the moment of crash, or
   b. were completed during the current DBRB before the moment of crash,

   are processed again. All these transactions are notified of the recovery if necessary.
5. Incoming transactions which were stored during recovery are processed.

In the case that errors that happened during a DBRB are not detected by Integrity Assertion Verifier at the end of this DBRB, a mechanism to restore earlier snapshots must be given. The mechanism is not a simple one by any means. If an error is detected for the ith time on end in the same DBRB during an attempted recovery, we can

1. try to run the DBRB one more time, assuming error transience or using one more permutation of transaction alternatives, or
2. back up to the previous snapshot and thus to the earlier DBRB.

With i growing, obviously the probability of the latter decision grows. But optimization of the decision is not easy.

We assume that the extent, the precision, and thus the cost of IA verification are much higher than those of any acceptance check. This relatively high cost is the reason that one cannot afford IA verification too often. Thus acceptance checks are still useful as means of earlier, specialized error detection. The costs of IA verification are nothing extravagant. T. Gibbons advises "It is wise to run a series of check programs on the database, to find all the errors before attempting a restart."[4]

Comparison of the performances of the "unsafe" and "safe" approaches under pessimistic circumstances shows the advantages of the latter. From now on we discuss the "safe" approach exclusively.

Assumptions for the Analysis of the Generic Technique for Database Recovery

We analyze the generic database recovery technique under the following assumptions:

A1. The database functions in a real-time environment.
A2. The database has a single division.
A3. Transactions are implemented accordingly to the recovery block scheme.
A4. Database recovery from the hidden hard crashes is considered.
A5. Integrity Assertion Verifier is completely effective (i.e. detects all errors). (At first sight this assumption seems to collide with our view of recovery-block acceptance tests as not completely effective. But there are important differences between the two:
   1. Integrity assertion verification is performed less often than acceptance test execution of any transaction. Thus integrity assertions can be more detailed and comprehensive with the comparable overhead.
   2. Integrity assertions are for general use, while acceptance tests are transaction-specific. Thus integrity assertions can be more thoroughly tested. Note that the assumption A5 could be discarded by a modification of our model as proposed above, namely by including a mechanism for the restoration of earlier snapshots when needed.)

A6. "Recovery" software is completely reliable. (Unlike the software of transactions, the "recovery" software, as a standard package, can be thoroughly tested and made quite reliable.)

Description of Database Recovery Techniques

The generic database recovery technique can be implemented in many ways. Our candidates are:[9,11]

1. Complete Database Dump—Before entry to each DBRB, the whole database is dumped (copied).
2. Incremental Dump—An initial or periodic database dump creates a basis. Before entering the next DBRB, all blocks/files updated in the previous DBRB are copied, i.e. incremental dump is created. This permits the restoration of the last snapshot, using the complete database dump, if necessary, and using the results recorded on incremental dumps. (Note that incremental dumps alone would not ensure recovery; blocks/files of the database not changed at all are not recorded on any incremental dump.)
3. Audit Trail—An audit trail (a log) records sequences of actions performed by transactions on files/blocks inside a given DBRB. It can be used to restore the latest snapshot. It can also be used to back up particular transactions, which is important when one needs to allow for abortion of a single transaction.
4. Differential Files—The main file (the frozen database) stores the latest snapshot, and the differential file is a log recording all later updates, executed inside of a DBRB. The merge of the differential file with the main file is done only after positive verification of the logical database made up by the main and differential files (i.e. before entering the next DBRB).
5. Backup/Current Versions—Copies blocks/files just before they are updated for the first time inside a DBRB. From then on only this copy of block/file is accessed. The "original" is a backup version used, if necessary, for database recovery. Using the latest backup copies for each block/file, the latest snapshot can be reconstructed.
6. Multiple Copies—More than one copy of each block/file is stored. The different copies are identical except during

an update. There are two variants of this technique. The first uses an odd number of copies and applies "majority voting" to select the correct data value. Fewer than half of the copies are ever updated at a time. The other variant uses only two copies, but each has an "update-in-progress" flag. A flag set indicates that the associated copy is under update and thus possibly in an inconsistent state. Only one copy at a time can be updated. Copies not under update at a moment of crash are consistent, if there are no hidden crashes.

7. Careful Replacement—The principle of this technique is the avoidance of updates "in place." Altered data are put in a copy of the original. The original is deleted only after the alteration is complete and has been certified. Note that two copies exist only during update.

Database Recovery Techniques— A Qualitative Analysis of Usefulness

Analyzing the potential usefulness of the presented database recovery techniques, we have found out that two of the techniques, multiple copies and careful replacement, can not be used for recovery from the hidden hard crashes. The multiple copies technique can be successfully used to recover from overt hard crashes or even, using majority voting, for error detection. But when hidden hard crashes occur, all copies could be equally contaminated and useless. The careful replacement technique deletes the original as soon as the new copy is certified. By definition, errors causing overt hard crashes are detected instantaneously and the technique can protect against them. But if hidden hard crashes occur and the IA verification is not completely effective (does not detect all integrity violations), the errors may be detected only some time after this verification. By then there is no way to restore the original, which has been deleted immediately after the IA verification.

Thus for the further analysis we are left with the following five database recovery techniques: complete database dump, incremental dump, audit trail, differential files, and backup/current versions.

Let us now try to answer the question: Which database recovery techniques could be used in the cases that (1) DBRB's are relatively short, (2) DBRB's are relatively long?

In the first case, clearly, we can afford undoing the results of database updates to back up to the most recent snapshot, so we do not need to prepare extensive physical database snapshots at the entry to a DBRB. Just logging the updated item values would suffice. Thus the audit trail technique seems suitable here.

In the second case, undoing the results of database updates would take too long. We must record database state (remember that we assume single-division database) at each DBRB entrance. The techniques that can be used here include

- complete database dump,
- incremental dump with an initial or periodic complete database dumps,
- differential files,
- backup/current versions.

Primitives for Database Recovery Techniques

We claim that it is both feasible and useful to present the database recovery techniques in terms of certain primitive actions, which we want to consider as atomic elements of the selected database recovery techniques. The feasibility is proved by the presentation of the set of these primitives, which follows.

Our long-term goal is the time-cost comparison of the database recovery techniques. Instead of analyzing each technique separately, we will analyze each primitive. As each recovery technique is a sequence of these primitives, the resulting recovery technique cost can be easily obtained. This is one of the aspects of the usefulness of the primitives. Others, we hope, will include the increased clarity of the description of these techniques.

Below we define the primitives and later we show how to construct the selected database recovery techniques out of these primitives.

In the definitions the notion of a set of "corresponding" pages, or of a "generic" page, is used: whenever page B of file Y was initialized as the copy of page A of file X, we say that these pages are corresponding or that both pages map into the same generic page, even if the content of page B, due to its updates, no longer is identical to the content of page A. In a sense, a generic page is the generalization, beyond a single file, of a page version. The function "pg" (as "page"), used in the figures for the next section, maps any file into the set of its generic pages. The function "gp[F]" maps a set of generic pages into the corresponding pages of a file F. (Note that the inverse of pg is not a function.)

The primitives are as follows:

C/DUMP(X)—Make complete database dump, and call it X.

COPY(Y,DB,X)—Copy all distinguished, i.e. with their IDs in X, pages of the database into the file Y (if a page has two versions in Y—delete the old one).

ERASE(X)—Erase block/file X.

HALT—Halt normal database processing after transactions currently writing into database write their results completely. This primitive ends a DBRB.

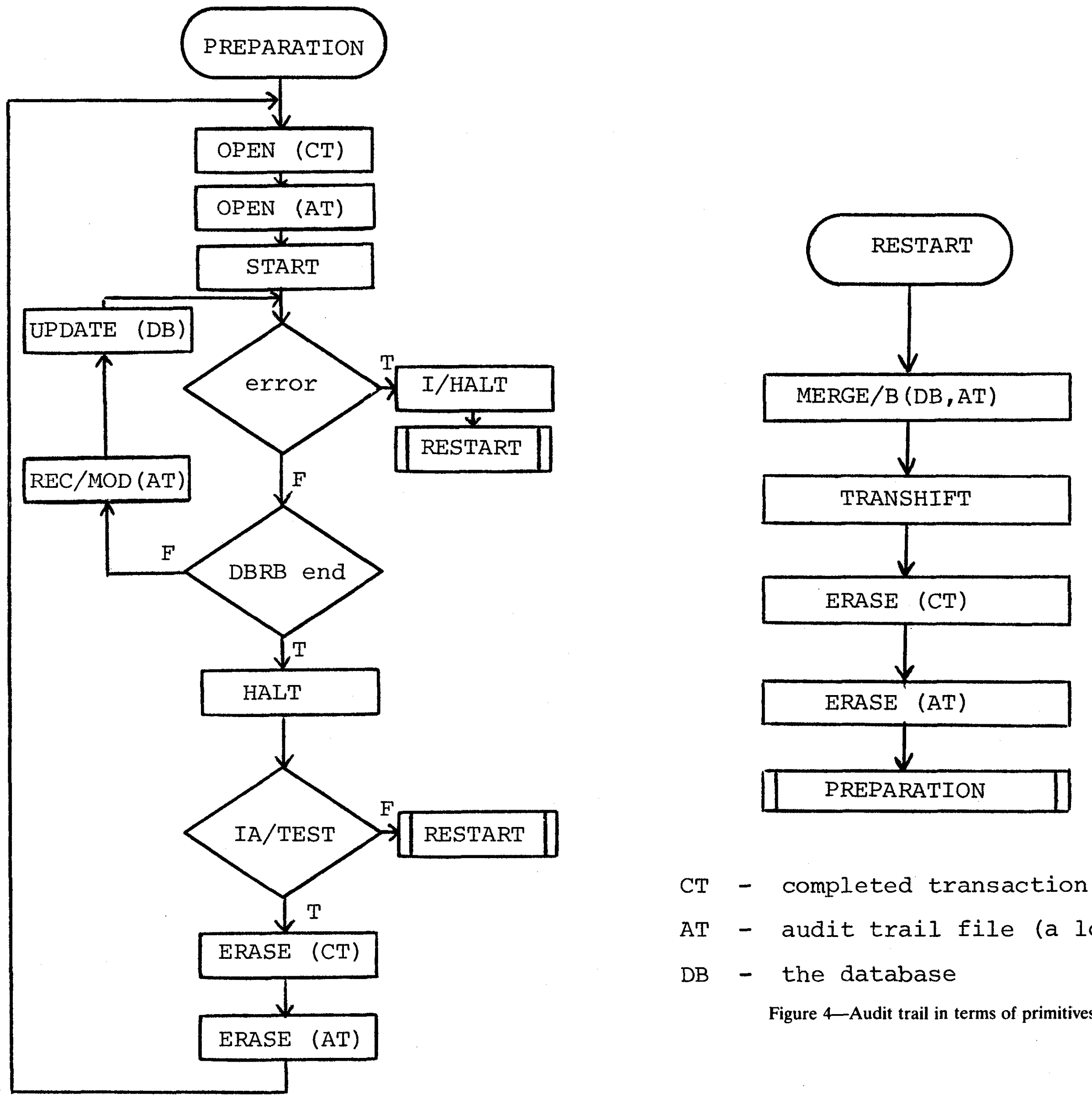
IA/TEST—Test original database consistency, using IA Verifier.

I/HALT—Halt normal database processing immediately upon detection of an error by an acceptance check. This primitive initiates restart of the current DBRB unconditionally, so we need not wait for writing transactions as in HALT.

LOG/IA/TEST(X)—Test, using the IA Verifier, the consistency of the current logical database. The current logical database consists of the most current values of database items that are stored in the database or in the block/file X. This corresponds to a logical merge of X with the database followed by IA/TEST.

MERGE/parameter(DB,X)—There are two variants:

1. *MERGE/B(DB,X)*—Merge the database with the log X backwards (i.e. use the oldest recorded values of data of X to restore the correct database).
2. *MERGE/F(DB,X)*—Merge the database with the log X



CT - completed transaction log

AT - audit trail file (a log)

DB - the database

Figure 4—Audit trail in terms of primitives

forward (i.e. use the newest recorded values of data of X to build the correct database).

OPEN(X)—Open file X.

OVERWRITE(X,Y,Z)—Replace (e.g., by pointer switching) pages of X specified by page identifiers stored in Z with the corresponding pages of Y. If Z is omitted—each page of Y replaces the corresponding page of X.

REC/ID(X)—Record in X identifiers of database pages to be modified.

REC/MOD(X)—Record data (e.g., a 4-tuple: transaction ID, item ID, old item value, new item value) about modifications on a log X.

START—Start normal database processing. This primitive initiates a new DBRB.

TRANSHIFT—Shift into system input queue:

a) all transactions recorded on "completed transaction log", i.e. finished but not saved transactions,
b) all other transactions present in the system, i.e. unfinished transactions,

and sort transactions of system input queue in the arrival time order.

UPDATE(X)—Write an update in the file X. This primitive specifies which file should be updated when more than one file includes the same generic page that is to be updated.

The Selected Database Recovery Techniques in Terms of Primitives

Using the primitives defined above, we have built the following selected recovery techniques:

1. audit trail (see Figure 4),
2. complete database dump (see Figure 5),

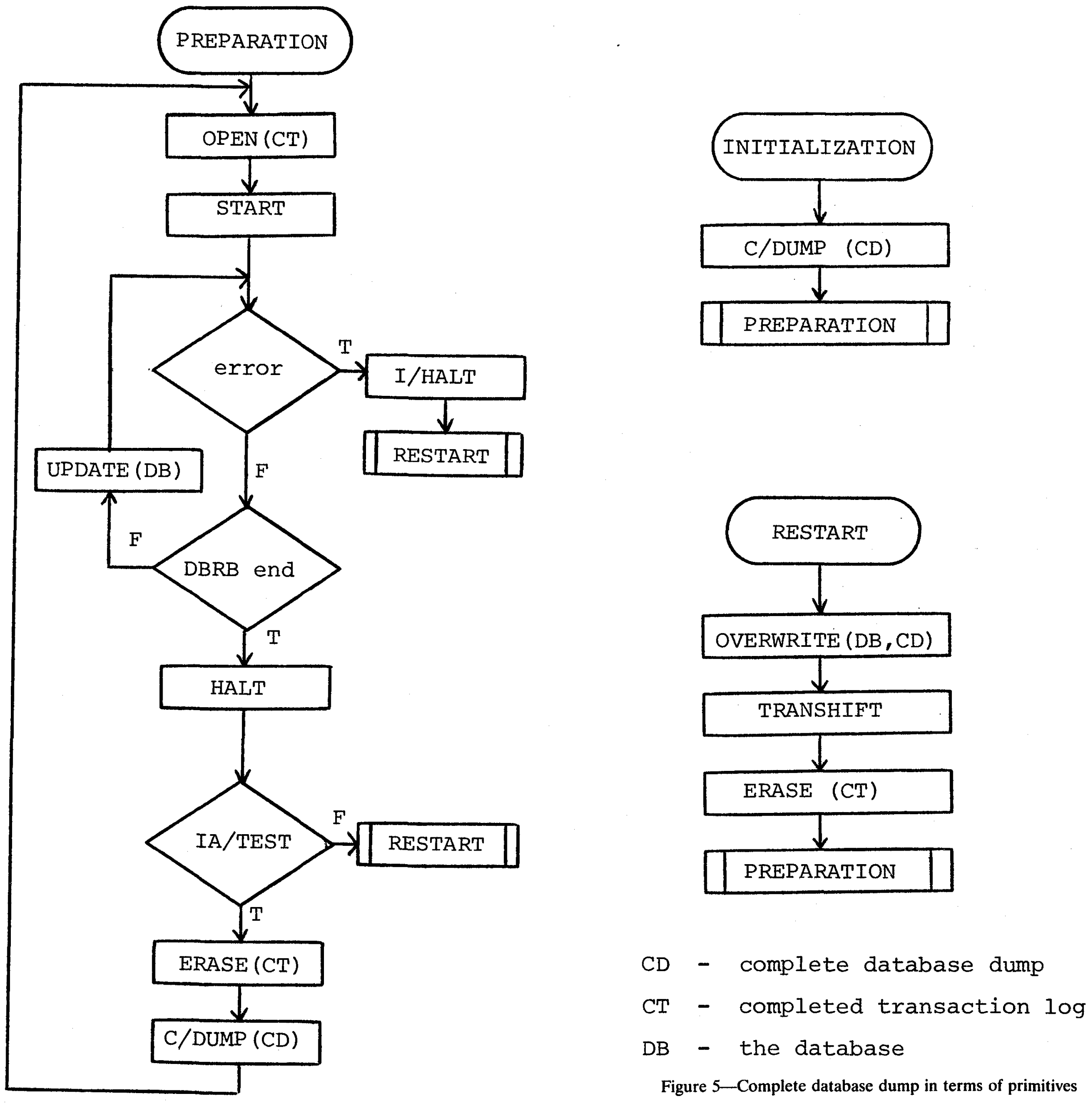


Figure 5—Complete database dump in terms of primitives

3. incremental dump (see Figure 6),
4. differential files (see Figure 7),
5. backup/current version (see Figure 8).

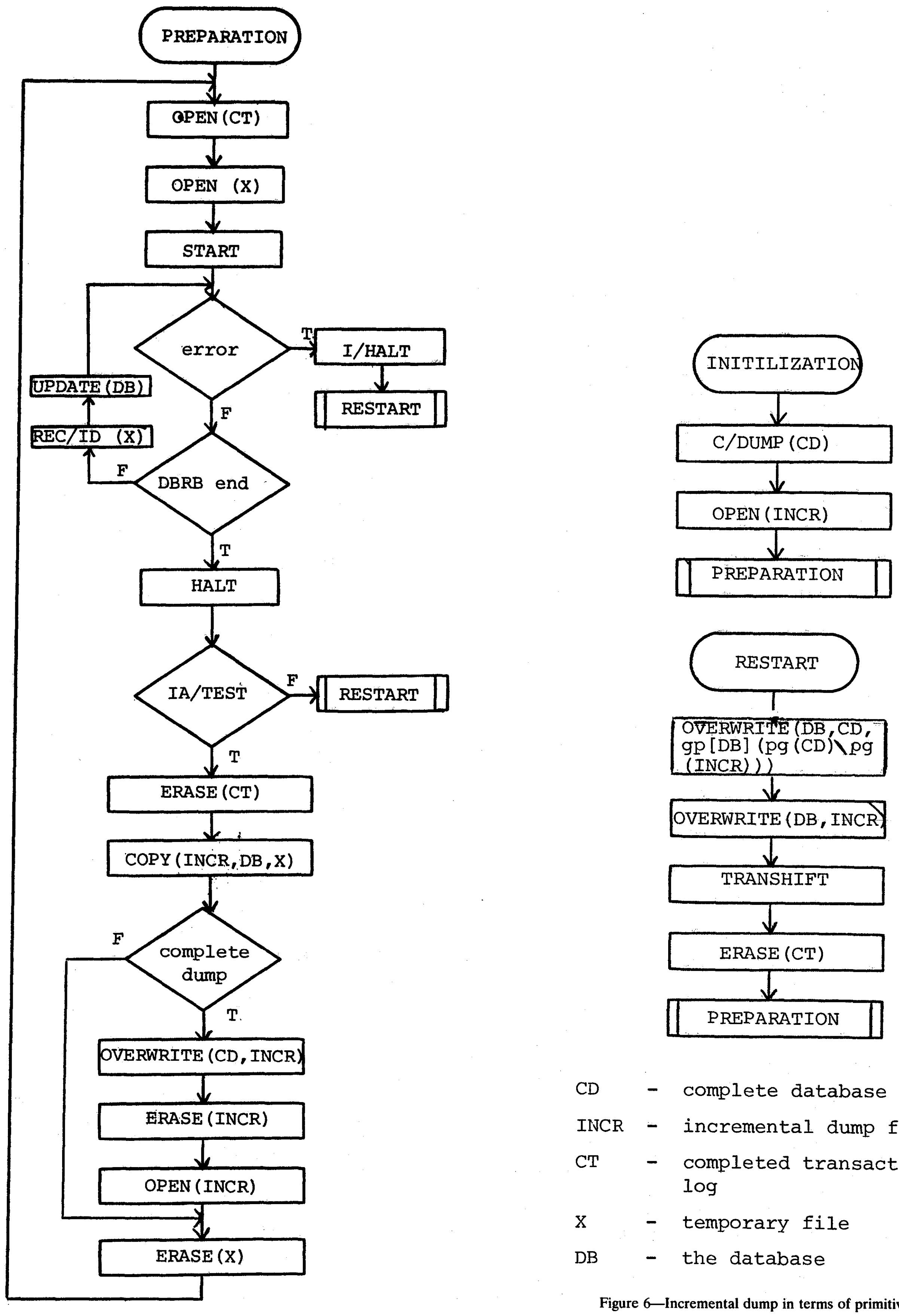
The flowcharts of these recovery techniques combined with the definitions of the primitives should be self-explanatory (you may wish to consult short description of the techniques in the section "Description of Database Recovery Techniques.") The completed transaction log, referred to in the above-mentioned figures, records all transactions that are completed (their results are already written into the database), but with updates not saved yet, that is, the end of the DBRB in which transaction finished its execution has not been reached. This allows it to reexecute completed transactions, if necessary.

For comparison we present in Figure 9 the list of the primitives used by the selected database recovery techniques. This demonstrates how much in common the techniques have.

FUTURE RESEARCH AND EXTENSIONS

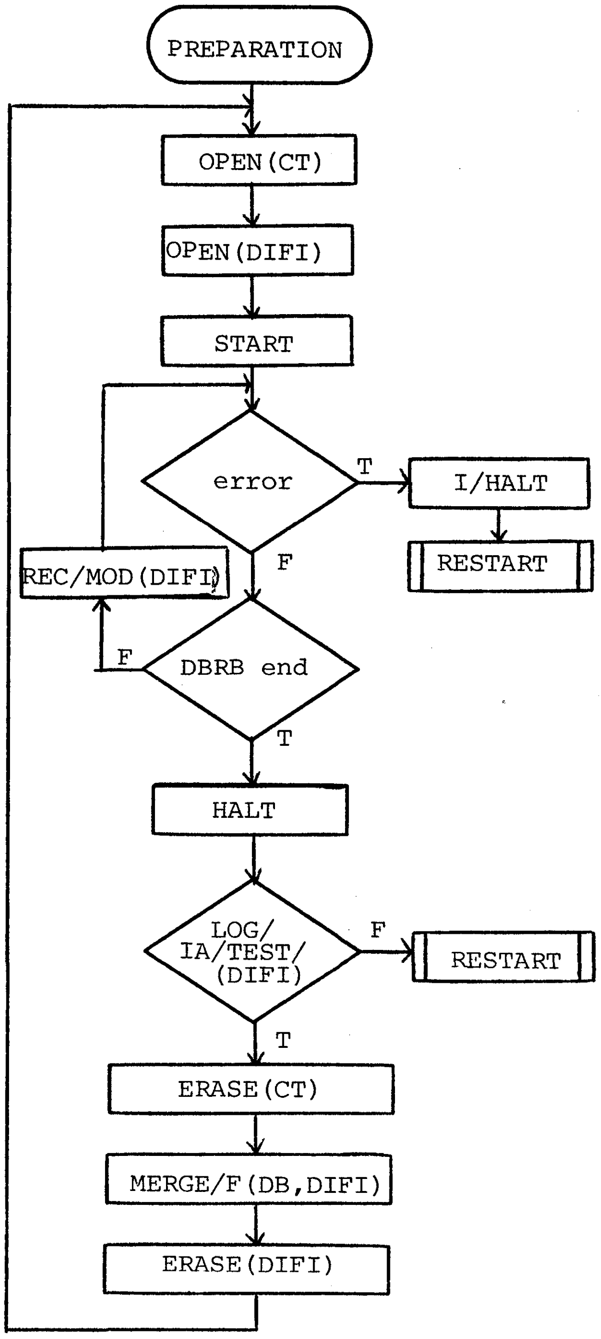
It is our intention to compare the performance of the above database recovery techniques for hidden hard crashes. We plan to base the analysis of the recovery techniques on the analysis of the primitives constituting them, which is to be made first.

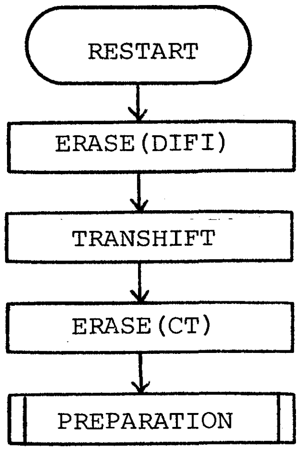
The database recovery cost considerations will be limited to the time-cost analysis, as the storage cost does not seem to be essential in the real-time environment. Time costs can be



CD	-	complete database dump
INCR	-	incremental dump file
CT	-	completed transactions log
X	-	temporary file
DB	-	the database

Figure 6—Incremental dump in terms of primitives



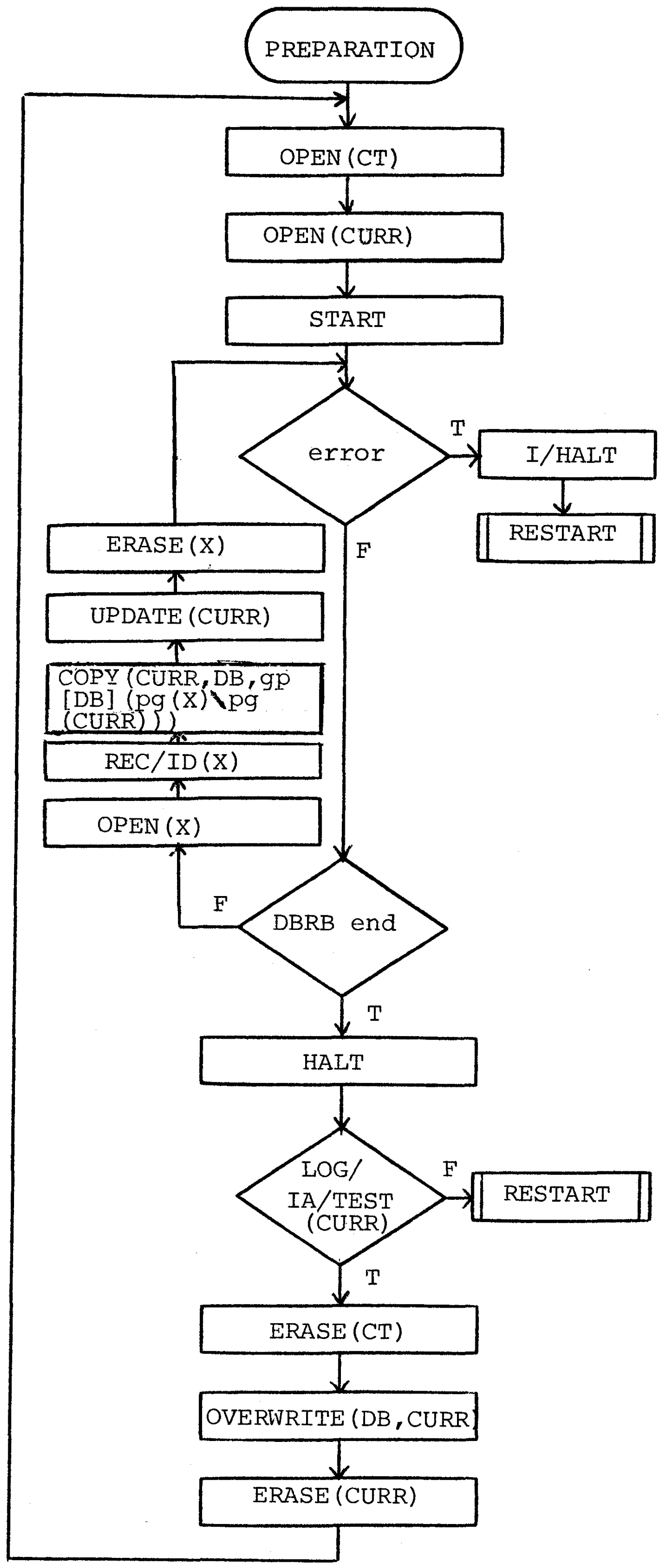


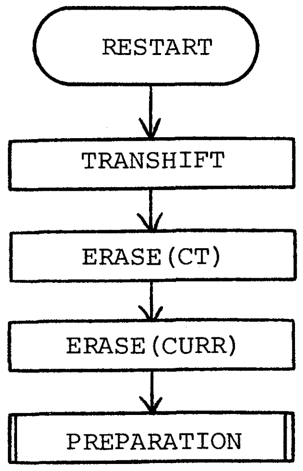
CT	-	completed transaction log
DIFI	-	differential file (a log)
DB	-	the database

Figure 7—Differential files in terms of primitives

classified as fixed and variable costs.[2] Fixed time costs, independent of the number of errors detected, cover all preparatory actions necessary for restart when an error is detected. Variable time costs, incurred only if an error is detected, cover all restart actions. The fixed time costs, as completely predictable, can be more easily incorporated within real-time constraints of the system operation during system design. But the variable time costs are the threat to real-time constraints of the system operation (these constraints could be defined as the maximum time the system can be left nonoperational without grave consequences). Thus in our opinion only the time-cost analysis is essential and the variable time cost is the main criterion of the cost analysis for a recovery technique in our environment.

A designer or a database administrator defines Database Recovery Blocks by specifying the intervals of regular database processing between consecutive recovery preparation phases. Long DBRB will increase chances that the restart will be time consuming, involving the reexecution of many transactions and keeping the system nonoperational too long. Short DBRB will increase the costs of the preparatory actions (snapshots, etc.), increasing the chances of breaking the real-time requirements. Thus a compromise is clearly needed. This compromise will affect operational costs of a given database





CT	-	completed transaction log
CURR	-	current version
DB	-	the database
X	-	temporary file

Figure 8—Backup/current version in terms of primitives

recovery technique. We want to find the minimum cost schedule for all of the above techniques.

Only the database recovery techniques for hidden hard crashes have been discussed. These techniques can obviously cope with the overt hard crashes too, but they are much more expensive than specialized recovery techniques. The tech-

PRIMITIVE \ RECOVERY TECHNIQUE	AT	CD	ID	DF	BC
C/DUMP(X)	-	IP	I	-	-
COPY(Y,DB,X)	-	-	P	-	P
ERASE(X)	PR	PR	PR	PR	PR
HALT	P	P	P	P	P
IA/TEST	P	P	P	-	-
I/HALT	P	P	P	P	P
LOG/IA/TEST(X)	-	-	-	P	P
MERGE/par(DB,X)	R	-	-	P	-
OPEN(X)	P	P	IP	P	P
OVERWRITE(X,Y,Z)	-	R	PR	-	P
REC/ID(X)	-	-	P	-	P
REC/MOD(X)	P	-	-	P	-
START	P	P	P	P	P
TRANSHIFT	R	R	R	R	R
UPDATE(X)	P	P	P	-	P

AT	-	Audit Trail
CD	-	Complete Database Dump
ID	-	Incremental Dump
DF	-	Differential Files
BC	-	Backup/Current Versions
I	-	Primitive used in initialization phase
P	-	Primitive used in preparation phase
R	-	Primitive used in restart phase
X,Y	-	File names
DB	-	The database

Figure 9—Use of the primitives by the selected database recovery techniques

niques for database recovery from overt hard crashes will be investigated later, using the analogous approach.

There are a number of possible extensions to our work:

1. increasing the concurrency of normal database processing by exploitation of elements of a recovery mechanism[1];
2. concurrent execution of recovery actions and normal database processing, for example, dumping concurrent with regular processing[4,6];
3. concurrent execution of a few recovery actions, such as checking database files concurrent with dumping of these files[4] or processing several logs (or log sections) in parallel (e.g., the Audit Trail Tag File method[4]);
4. creating single transaction backup facilities by use of deferred commit[4,7] or use of transaction save points;[6]
5. independent dumping of sections of a database, especially when these sections have varying level of activity or the database is large (compare the noncontemporary file dumps method[4]);
6. investigation of special database recovery implementation methods, for example the duplexing of logs and files[4,7] or the use of multiprocessor systems; and
7. investigation of after-implementation tunability of recovery methods.

In the refinement of our approach we will include some of these ideas.

ACKNOWLEDGMENT

We would like to thank Ms. Beverly Hill for the preparation of the figures.

REFERENCES

1. Bayer, R., H. Heller, and A. Reiser, "Parallelism and Recovery in Database Systems," *ACM Trans. on Database Systems,* June 1980.
2. Chandy, K.M., J.C. Browne, C.W. Dissly, and W.R. Uhrig, "Analytic Models for Rollback and Recovery Strategies in Data Base Systems," *IEEE Trans. on Software Engineering,* March 1975.
3. Garcia-Molina, H. *Reliability Issues for Completely Replicated Distributed Databases.* Princeton University, Dept. of EECS, Technical Report #266, 1980.
4. Gibbons, T. *Integrity and Recovery.* Hayden Book Company, 1976.
5. Giordano, N.J., and M.S. Schwartz, "Data Base Recovery at CMIC," 1976 SIGMOD International Conference on Management of Data.
6. Gray, J., P. McJones, M. Blasgen, et. al. *The Recovery Manager of a Data Management System.* IBM Technical Report RJ 2623.
7. Gray, J. *A Transaction Model.* IBM Technical Report, February 1980.
8. Randell, B., "System Structure for Software Fault Tolerance," *IEEE Trans. on Software Engineering,* June 1975.
9. Sayani, H.H., "Restart and Recovery in a Transaction-Oriented Information Processing System," *ACM SIGMOD Workshop on Data Description, Access and Control,* 1974.
10. Sibley, E.H., "The Development of Data-Base Technology," *Comp. Surv.,* March 1976.
11. Verhofstad, J.S.M., "Recovery Techniques for Database Systems," *Comp. Surv.,* June 1978.