Disaster recovery techniques for database systems.

ARTICLE *in* COMMUNICATIONS OF THE ACM · JANUARY 2000

Impact Factor: 3.62 · DOI: 10.1145/352515.352521 · Source: DBLP

Recovery Techniques For Database Systems*

JOOST S. M. VERHOFSTAD

BNSR, 522 University Avenue, Toronto, Ontario M5G 1W7 Canada

A survey of techniques and tools used in filing systems, database systems, and operating systems for recovery, backing out, restart, the maintenance of consistency, and for the provision of crash resistance is given.

A particular view on the use of recovery techniques in a database system and a categorization of different kinds of recovery and recovery techniques and basic principles are presented. The purposes for which these recovery techniques can be used are described Each recovery technique is illustrated by examples of its application in existing systems described in the literature.

A main conclusion from this survey is that the recovery techniques described are all useful; they are applied for different purposes and in different environments. However, a certain trend in the increasing use of specific techniques during the past few years can be noted. Another main conclusion is that there are still enormous integrity and recovery problems to be solved for parallel processes and distributed processing.

Keywords and Phrases: audit trail, backup, checkpoint, database, differential files, filing system, incremental dumping, recovery.

CR Categories: 1.3, 3.73, 4.33, 4.9

INTRODUCTION

Recovery techniques can be used to restore data in a system to a usable state. Such techniques are widely used in filing systems and database systems in order to cope with failures. A failure is an event at which the system does not perform according to specifications. Some failures are caused by hardware faults (e.g., a power failure or disk failure), software faults (e.g., bugs in programs or invalid data), or human errors (e.g., the operator mounts a wrong tape on a drive, or a user does something unintentional). A failure occurs when an erroneous state of the system is processed by algorithms of the system. The term *error* is, in this context, used for that part of the state which is "incorrect." An error is thus a piece of information which can cause a failure [MELL77].

In order to cope with failures, additional components and abnormal algorithms can be added to a system. These components and algorithms attempt to ensure that occurrences of erroneous states do not result in later system failures; ideally, they remove these errors and restore them to "correct" states from which normal processing can continue. These additional components and abnormal algorithms, called *recovery techniques,* are the subject of this survey.

There are many kinds of failures and therefore many kinds of recovery. There is

* The research work described here was done while the author was at the Computing Laboratory, Claremont Tower, Claremont Road, University of Newcastle upon Tyne, UK.

CONTENTS

INTRODUCTION
OVERVIEW OF RECOVERY TECHNIQUES
SALVATION PROGRAMS
INCREMENTAL DUMPING
AUDIT TRAIL
DIFFERENTIAL FILES
BACKUP AND CURRENT VERSIONS
MULTIPLE COPIES
CAREFUL REPLACEMENT
DIRECTIONS FOR FUTURE RESEARCH AND RELATED WORK
SUMMARY AND CONCLUSIONS
ACKNOWLEDGMENTS
REFERENCES

always a limit to the kind of recovery that can be provided. If a failure not only corrupts the ordinary data, but also the *recovery data*—redundant data maintained to make recovery possible—complete recovery may be impossible. As described by Randell [RAND78], a recovery mechanism will only cope with certain failures. It may not cope with failures, for example, that are rare, that have not been thought of, that have no effects, or that would be too expensive to recover from. For example, a head crash on disk may destroy not only the data but also the recovery data. It would therefore be preferable to maintain the recovery data on a separate device. However, there are other failures which may affect the separate device as well—for example, failures in the machinery that writes the recovery data to that storage device.

Recovery data can itself be protected from failures by yet further recovery data which allow restoration of the primary recovery data in the event of its corruption. This progression could go on indefinitely. In practice, of course, there must be reliance on some ultimate recovery data (or rather, acceptance that such recovery data cannot be not totally reliable).

Techniques and utilities that can be used for recovery, crash resistance, and maintaining consistency after a crash are described in this survey. Included are descriptions of how data structures should be constructed and updated, and how redundancy should be retained to provide recovery facilities. This survey deals with recovery for data structures and databases, not with other issues that are also important when processes operate on data, such as locking, security, and protection [LIND76].

One possible approach to recovery is to distinguish different kinds of failures based on two criteria: 1) the extent of the failure, and 2) the cause of the failure. The three kinds of failures typically distinguished in many database systems are: a failure of a program or transaction; a failure of the total system; or a hardware failure. Different "recovery procedures" are then used to cope with the different failures. A good description of such an approach to recovery has been given by Gray [GRAY77].

However, recovery is approached from a different angle in this paper. To be able to recover, two kinds (i.e., functionally different kinds) of data are distinguished: 1) data structures to keep the current values and 2) recovery data to make the restoration of previous values possible. This paper examines:

- how these data structures and recovery data can be structured, organized and manipulated to make recovery possible (all this is referred to as the *recovery technique*);
- how the data structure can interact with the structure of the recovery data;
- what kinds of failures can be coped with by the different organizations;
- what kind of recovery (e.g., restoration of the state at time of failure, the previous state, and so on) can be provided using these organizations;
- and how different techniques can be combined in one system to cope with different failures or to provide different kinds of recovery (e.g., one technique may be used as a fall back for another one).

The first approach used in most commercially available data base systems is based on the requirement that the system be able to undo, redo, or complete transactions. A *transaction* is the unit of locking and recovery; it therefore appears to the user as an atomic action. The recovery techniques considered in this paper can be used to implement this atomic property for user actions on the database, but the techniques are described from the data structuring point of view.

For the purpose of this survey the notions of filing system and database system are treated as synonymous. The definition of the notion of *database* given by Martin [MART76] is used here: A database is a collection of related storage objects together with controlled redundancy to serve one or more applications; the data in the database is stored so as to be independent of programs using them; a single approach is used to add, modify, or retrieve in the database.

A database may consist of a number of files. A *file* is a logical unit in the database, used to group data. The data that can be retrieved by users from the database forms the *information* in the database. Thus, if some of the data stored in the database becomes irretrievable, some information is lost.

The concepts of database, file, and information are logical. The physical database is held in secondary storage. *Secondary storage* is nonvolatile storage space, which retains the database whether or not it is online (mounted on a storage device unit and readable by computer). Secondary storage consists of *physical records,* which are the smallest accessible units. Records are read or written by a storage device unit at the request of the computer.

A database is an abstraction of secondary storage provided to the user by the *database system.* The database system implements the user operations on the database and implements the data structures on secondary storage. *Objects* are the substructures from which these data structures are built. Examples of objects are: a logical record, a header of a file, a linked list of pages or logical records, and an entry in a directory. Objects are mapped onto records which comprise secondary storage.

Users may add, delete, and update data. The database is in a *correct state* if the information in it consists of the most recent copies of data put in the database by users and contains no data deleted by users. A database is in a *valid state* if its information is part of the information in a correct state. This implies that there are no spurious data, although some information may have been lost. A database is in a *consistent state* if it is in a valid state, and the information it holds satisfies the users' consistency constraints.

It is assumed here that a correct state is also a consistent state. For example, if a machine is suddenly halted, and a database state is defined at that time, then this state is called the correct state. If no state is defined for the database when the machine is halted, a salvager can be run to delete parts of the database in order to restore the system to a defined state: a valid state. If this salvager also makes sure that user's consistency constraints are maintained, then the restored state is a consistent state.

Consistency will have to be a well-defined notion for every database. Different sorts of consistencies (possibly at different levels of abstraction) or degrees of consistencies [GRAY76] may be defined. No more precise definition of consistency will be given here.

To illustrate these definitions with another example, consider a user who maintains a source file, and an object file which has been produced by compiling the source file. The database will then be in a correct state if the most recent source and object files are available. The database will be in a valid state if a source file and an object file, but not necessarily the most recent ones, are available. The database will be in a consistent state only if corresponding source and object files are available.

A *failure* of the system occurs when that system does not meet its specifications. *Recovery* is the restoration of the database after a failure to a state that is acceptable to the users. The notion of "acceptable" is different for different environments; in general, "acceptable" will mean correct, valid or consistent. A *recovery technique* provides recovery from certain kinds of failures. Within a single system, there may be several different recovery techniques corresponding to different kinds of failures. However, it is common to structure the techniques into a hierarchy; the most general ones deal with the largest set of failures, but are the most expensive. The recovery technique for the smallest set of failures is usually the most efficient technique and involves minimal loss of information. An example of this is given below and illustrated in Figure 1.

A recovery technique maintains *recovery data* to make recovery possible. It provides recovery from any failure which does not affect the recovery data or the mechanisms used to maintain these data and to restore the states of the data in the database. Failures are classified into two groups with respect to a recovery technique. A failure with which a recovery technique can cope

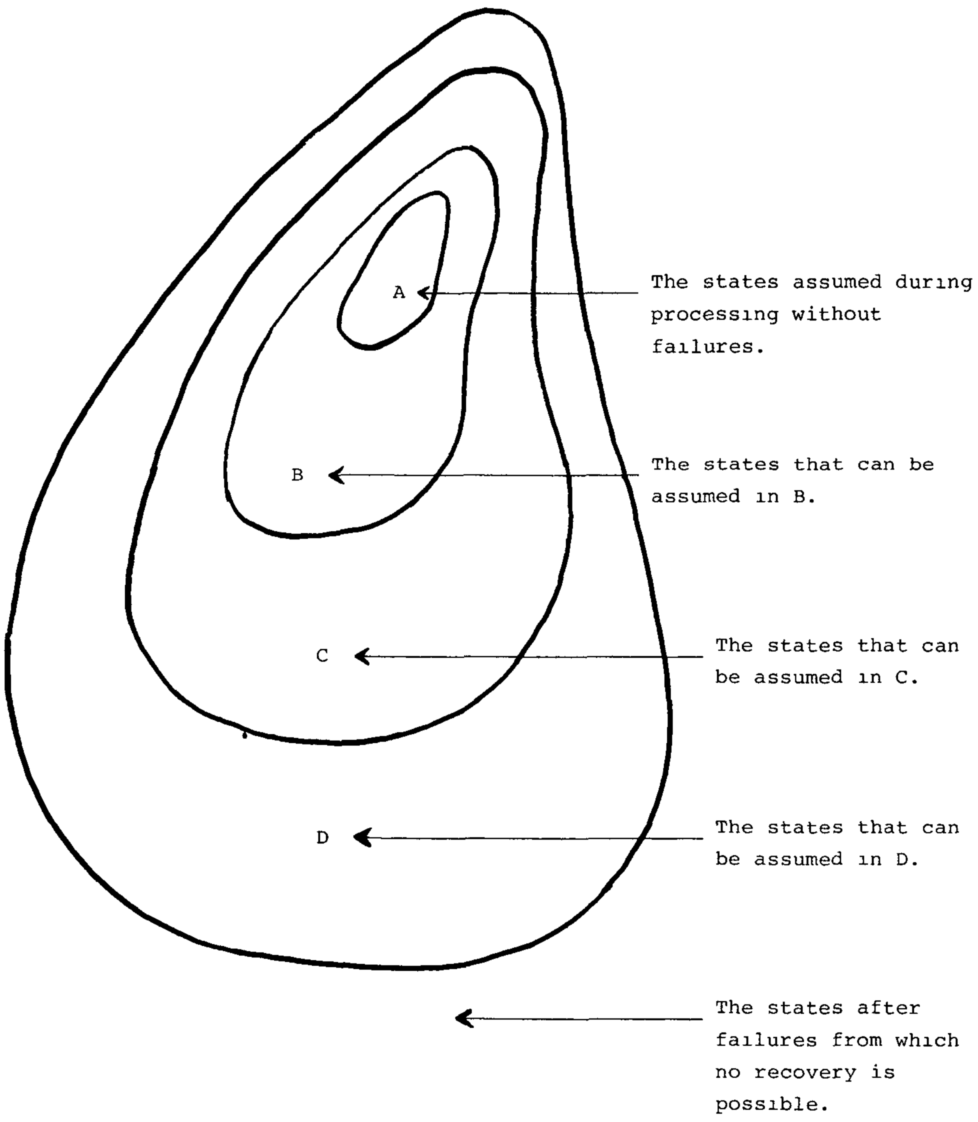

A
The states assumed durıng processıng without faılures.
B
The states that can be assumed ın B.
C
The states that can be assumed ın C.
D
The states that can be assumed ın D.
The states after faılures from whıch no recovery is possıble.

FIGURE 1. The state space for a database system wıth several recovery technıques, copıng with subsequently larger sets of faılures.

is a *crash* of the system with respect to that recovery technique. A failure with which a recovery technique cannot cope is called a *catastrophe* with respect to that technique.

A system using three recovery techniques could, for example, consist of the following subsystems (see also Figure 1):

A) The database system without any recovery techniques.
B) "A" plus a recovery technique that uses built-in redundant pointers in data structures to be able to recover from certain failures causing particular errors in the data structures.
C) "B" plus a recovery technique that does not use built-in redundancy in data structures, but maintains backup copies of (parts of) the data structures.
D) "C" plus a recovery technique that keeps a complete backup copy of the database on a separate device.

These systems could be built using an approach similar to the "safe-programming" approach described by Anderson [ANDE75]. The bigger the damage, the cruder the recovery technique used. Restoration of the correct state is most desirable and can be done, say, in B. However, if the damage is such that recovery in B is not possible, then restoration to a consistent, but not necessarily the correct, state may be the only alternative in C.

No single recovery technique or series of recovery techniques can cope with every possible failure. Many different kinds of recovery procedures have been developed, each technique with its own particular advantages and disadvantages, but each enabling the system to cope with different kinds of failures in different environments.

In the following sections the categories of recovery techniques known and used at present are briefly described; the kinds of recoveries they provide and the relationships among the techniques are given. Next, the different techniques are defined and described in detail, along with a consideration of the purposes for which they can be employed, and of the systems which use them. Finally, some conclusions are drawn and some recent trends are described.

OVERVIEW OF RECOVERY TECHNIQUES

Different kinds of recovery are possible for a database. The "kinds of recovery" that we consider are in fact "qualities of recovery," which can be useful in comparing and evaluating different recovery techniques. The kinds of recovery considered are:

1) Recovery to the correct state.
2) Recovery to a correct state which existed at some moment in the past (i.e., a checkpoint).
3) Recovery to a possible previous state; this would allow, for example, restoration of a set of previously existing states of files that may not have existed simultaneously before.
4) Recovery to a valid state.
5) Recovery to a consistent state.
6) Crash resistance (explained below).

Crash resistance is provided if the normal algorithms of the system operate on the data in such a manner that after certain failures the system will always be in a correct state, i.e., the state the system was in before the last operation on the data was started (or possibly the last series of operations). Thus, crash resistance obviates the need for recovery techniques to cope with a certain class of failures.

Crash resistance differs from other kinds of recovery. Whereas other kinds of recovery explicitly restore states, crash resistance maintains correct states by the way data are manipulated and maintained during normal processing. Thus, in a sense, crash resistance restores states implicitly. These differences are fully explained later in this survey. The notion of crash resistance cannot be made more precise than the notion of consistency, for the two are related. However, it is not necessary to be more precise to achieve the objectives of this survey.

A *checkpoint* is a (presumably correct) past state; it may have been made by recording the past state explicitly. Checkpoints are used by recovery techniques of kinds 1, 2 and 3 (but not necessarily 4 and 5, see definitions). Checkpoints can be established either for files or for the whole database. The creation of a checkpoint is called *checkpointing*. Establishing a checkpoint explicitly creates a *backup version,* which is a complete copy of the checkpointed file (or database). The term *backing up* means restoring the state of the previous checkpoint.

"Backing up" should not be confused with a different term, "backing out." The term *backing out* is related to processes or transactions. A process is backed out if all the effects of the operations performed by that process are undone. This means that only the files affected by the process need to be restored. Backing out of some processes may be required, for example, to resolve a deadlock or to undo the operations of a failing process. Backing out is a special sort of recovery of kind 3: only the data affected by the programs that are backed out are restored. So the total database is restored to a state which has been termed a "possible previously existing state."

For the purpose of this paper, techniques used for recovery, restart, and maintenance of consistency are divided into seven categories. (This categorization is, of course, not the only possible one.) The remainder of the survey deals with these seven recovery techniques. Systems described in the computer literature are used to illustrate how the recovery techniques have been implemented. Some of the systems described may have been changed over the past few years, so the descriptions of all systems may not be up to date anymore. However, the purpose of the examples is to illustrate how the techniques have been used, rather than to give accurate up-to-date descriptions of actual systems.

1) **Salvation program**——A salvation program is run after a crash to restore the system to a valid state. It uses no recovery data. (It is the only technique considered here which does not use recovery data.) It is used after a crash if other recovery techniques (using recovery data) fail or are not used, or if no crash resistance is provided. This program scans the database after a crash to assess the damage and to restore the database to some valid state. It rescues the information that is still recognizable.
2) **Incremental dumping**——Incremental dumping involves the copying of updated files onto archival storage (usually tape) after a job has finished or at regular intervals. It creates checkpoints for updated files. Backup copies of files can be restored after a crash.
3) **Audit trail**——An audit trail records sequences of actions on files. It can be used to restore files to their states prior to a crash or to back out particular processes. It can also be used for certifying that rules and laws are obeyed in the system. An audit trail provides the means to back out a process whereas incremental dumping merely provides the means to restore files to previous consistent states.
4) **Differential files**——A file can consist of two parts: the *main file* which is unchanged, and the *differential file* which records all the alterations requested for the main file. The main files are regularly merged with the differential files, thereby emptying the differential files. Records in the differential files can be stored with the process identifier, a time stamp, and other identification information to provide such special facilities as auditing, recovery, or crash resistance. A differential file is a type of audit trail, but the actual updates have not yet occurred. The differential file can also be used to implement crash resistance.
5) **Backup/current version**——The files containing the present values of existing files form the current version of the database. Files containing previous values form a consistent, backup version of the database. Backup versions can be used to restore files to previous values.
6) **Multiple copies**——More than one copy of each file is held. The different copies are identical except during update. A "lock bit" can be used to protect a file during updating, while its state is inconsistent. If there is an odd number of files, comparison can be done to select a consistent version. This technique provides crash resist-

ance; it may be used to detect faults if the different copies are kept on different devices or handled by different processors.

The difference between multiple copies and backup/current version is like the difference between TMR and standby: with multiple copies all copies are active, while with backup/current version there is only one active copy (other copies could even be off-line).

7) **Careful replacement**——The principle of the careful replacement scheme avoids updating any part of a data structure "in place." Altered parts are put in a copy of the original; the original is deleted only after the alteration is complete and has been certified. The difference between this and the other methods is that two copies exist *only* during update. The technique is used to provide crash resistance, for the original will always be available in case a crash occurs during update.

A cross-reference table between the categories of recovery and the recovery techniques is given in Figure 2. Strictly speaking, the table is incomplete because, for example, an audit trail or differential files can be used to restore a valid state. However, missing cross-references indicate techniques that would never be used for those purposes since one can always do better (e.g., restore the correct state rather than a valid state).

From the description of the techniques in Figure 2 the following relationships between the techniques are apparent:

- The differential file technique makes incremental dumping very easy to implement. Incremental dumping, in general, copes with failures that the differential file technique cannot handle (this is not apparent from Figure 2). However, the kinds of recovery provided by the differential file techniques are preferable to those provided by incremental dumping (as shown in Figure 2). Thus, the two techniques may complement each other very well.
- The audit trail technique is an alternative to differential files, careful replacement, or multiple copies; it can be used to restore the correct state after a crash. It is therefore seldom used in practice in conjunction with these other methods, although it may be used in combination with one of them in order to provide (the same kind of) recovery from different failures. The audit trail can thus be used to provide recovery from the failures for which the other techniques are also used, even though audit trail may be less efficient.
- Multiple copies and careful replacement may be used either as alternatives or as complements which provide crash resistance against similar types of failures. (We will return to this shortly.)
- Also the incremental dumping, the audit trail, the differential files, and the backup/current version techniques can be used as alternative techniques to provide recovery from particular fail-

	1) *Correct State*	2) *Previous State*	3) *Pos. Prev State*	4) *Valid State*	5) *Consistent State*	6) *Crash Resistence*
Salvation Program				*	*	
Incremental Dumping			*	*		
Audit Trail	*	*	*			
Diff. Files		*	*			*
Backup Current		*	*			
Multiple Copies						*
Careful Replacement		*				*

FIGURE 2. A cross-reference table indicating for what purposes the various recovery techniques can be used

ures or to complement each other to provide (the same kind of) recovery from different failures.

- The salvation program as a recovery technique is a last resort, used if all other techniques fail. It cannot bring the database back to a previous state. It merely rescues what is left. However, a salvation program can be used as a recovery technique for recovery data rather than for the database. For example, a salvation program can be used to restore the audit trail immediately after a failure of the system. The restored audit trail can then be used to restore the database. In this case the salvation program is used rather early.

The seven techniques under discussion provide recovery, crash resistance, and maintenance of consistency in one of three ways:

- The way in which the data is structured. The multiple copies, differential files and backup techniques are part of the structure of the database.
- The way in which the data is updated and manipulated. The careful replacement technique is a crash-resistant way of updating complex data structures. It has been shown [VERH77b] that this also sets special constraints and requirements for the data structures.
- The provision of utilities. The salvation program, incremental dumper and audit trail facility are utilities which have nothing to do with the way in which the data is structured or updated. They could be regarded as external utilities which can usually be added to any database system without great difficulty.

Unfortunately, this division of the techniques into three groups is too coarse: it can be misleading in cases where different techniques in one group complement each other or different techniques from different groups are alternatives. The seven techniques are therefore discussed separately, and examples are given of systems on which they are implemented.

SALVATION PROGRAMS

A salvation program in a database system is used after a crash to restore the database to some consistent state. The salvation program tries to restore the state of the database as it was before or at the time of the failure. However, some files or data may be lost. A salvation program scans through the data structures and tries to reconstruct the database or restore consistency, possibly at the cost of deleting some files or data.

A salvation program is needed after a crash if the data kept on secondary storage is not kept in a consistent state all the time, or if no other recovery technique is available to cope with the failure. Otherwise there is no need for such a program.

One reason the data on secondary storage might be inconsistent after a crash would be the loss of buffers kept in main storage. Some inconsistent files may have to be deleted because of [SMIT72]: violation of standard error checks on reading a file; conflict resulting from the same storage having been assigned to more than one file; or conflict (e.g., on the file length) determined from redundant information (e.g., from a file header).

A system may use data buffers (for the database) and audit buffers (for audit tapes). After a crash there may be no way to tell which updates recorded in the audit trail have been written to the database and which were still in data buffers; there may be no way to tell which successful updates were recorded on audit trail tape or were still in audit buffers at the time of the crash. Thus, the audit trail may not succeed in restoring the database to its state at the point of failure.

Several systems such as IMS [IBM] or the CMIC system [GIOR76], when running on machines using core storage, first use a salvation program which tries to rescue the contents of the buffers in main storage in order to close the audit trail tapes properly. However, if the contents of main storage are lost, restoration of the correct state is not possible.

At present LSI memories are widely used. However, the contents of LSI memories generally do not survive power failures. IMS therefore has implemented, in the last few years, the "log tape write ahead protocol." This procedure forces the audit trail to be written before the object in the database is written. Thus, buffers can still be used as long as the system conforms to the "protocol."

A system in which a salvation program is of great importance is the HIVE system [TAYL76] (here the program is called the recovery procedure). The system consists of a fixed number of virtual processors (VPs) which are assigned permanently to execute cyclically particular functional application programs. A processor cycle, performing such a particular function, is triggered by a message received from another VP or from outside the network of VPs. Capabilities (descriptions for resources available) for the necessary code and permanent data areas are given to the VPs at system build time, and the message routes between VPs are also set up permanently.

Only the files for which permanent capabilities have been created at system-initialization can be restored after a crash. Thus, a possible previous state of the system is restored: the files that existed at system initialization time are restored to the states they were in before the current transactions started. Also:

- transaction checkpoints can be made by writing the data of each transaction into a common, permanent, safeguarded checkpoint file, which can be accessed and recovered after a crash; and
- files may be created dynamically and capabilities for them may be put in special files called cap-files.

The recovery procedure run after a crash restores in main memory the read-only image, which also contains recovery code. The main task of the recovery procedure is a garbage collection by scanning all files in the database. Files whose capabilities are kept in cap-files are processed first. These files were created after system initialization and can be restored (using the cap-files) to the states they were in before the current transactions were started. The system is thus restored to a possible previous state which is an enhancement of the possible previous state that could be restored without the use of the cap-files.

This state can be yet further enhanced by the use of checkpoint files. The checkpoint files can be used to restore files to a checkpoint state; thus, the processing of transactions can be restarted from checkpoints. For each version of each file (several versions of each file are maintained) the check sums are evaluated to detect partially updated and corrupted pages; where possible, the appropriate updating and backtracking from other versions is carried out. (Only one version can be corrupted during a crash because different versions were kept on different disks, and only one version is updated during a transaction. The other versions are updated only after transactions are completed and the updated version is in a new correct state. A corrupted state can be detected using the check sums. Only a catastrophe, such as a fire in the computer center, could corrupt more than one version.)

Other systems in which a salvation program is used to recover the disk contents after system failure have been described, for example, by Lockemann and Knutsen [LOCK68], Daley and Neumann [DALE65] (salvage procedure), Fraser [FRAS69] (start-up procedure), and EMAS [EMAS 74]. (See also the surveys in [TONI75] and [MASC73].)

INCREMENTAL DUMPING

Incremental dumping is used to copy updated files onto archive storage (usually tape); it checkpoints files that have been altered. Incremental dumping is normally done after a job is finished, but can also be done at regular intervals, while continued use is made of the files, thereby providing more frequent checkpoints. After a crash has occurred the incremental dump tapes can be used to bring all the files to their previous consistent state, so that jobs completed before the crash will not be lost. All updates performed by jobs running at the time of the crash may not be restored completely by the processing of the incremental dump tape after a crash, because some active files may not have been dumped in time.

Fraser [FRAS69] gives a very good description of the technique used at Cambridge, which makes complete copies of updated disk files every 20 minutes. (See also [WILK75].)

In the original MULTICS system [DALE 65] all disk files updated or created by the user are copied when the user signs off. All newly created or modified files, which have not previously been dumped, are also copied to tapes once per hour. The original MULTICS scheme provides high reliability. However, overheads in resources and processing time are far too high; recovery time after failure is too high; and the system must be shut down periodically for backup purposes. It is most discouraging that the situation steadily worsens with system growth.

The design of a greatly improved backup mechanism has been described by Stern [STER74]. It is based upon the original backup mechanisms contained in the MULTICS system. Compared with the original system, this new design lessens overhead, drastically reduces recovery time from system failures, eliminates the need to interrupt system operation for backup purposes, and scales up significantly better with online storage growth. Some of the major features of this new scheme are:

- Incremental dumping is used to keep the backup system up to date.
- A complete secondary dump supersedes the complete incremental dumping history of the system: Rather than dumping the entire secondary storage, the procedure updates a checkpoint of the total system. A partial secondary dump supersedes part of the incremental dumping history. Secondary dumps are similar to change accumulation sets in IMS [GRAY77].
- A shadow copy of a file can be created to make sure that the incremental dumper dumps a consistent version.
- Each storage device holds a complete subtree or several subtrees of the file hierarchy (the MULTICS file system is organized as a tree of files [DALE 65]). This minimizes the effects of the loss of one device.
- After a failure a "salvager" is used to correct, detect, and report wherever possible any inconsistencies in the file hierarchy and storage tables. The system can be made available immediately after this; some files may still need to be reloaded, but they are marked as such. Directories and system files are reloaded first in order to make the system available to users again as quickly as possible. There are parallel processing capabilities for the reloading operation; and no unnecessary searches on dump tapes are made by the reloader. Only missing files are reloaded.
- The dumper can avoid unnecessary searching in the tree because of the use of dates and times in the directories in the tree. Also the reloader avoids unnecessary searches and file reloads.

The currently used recovery techniques in MULTICS are similar to those described by Stern. However, since the publication of Stern's thesis a major change has been made to the MULTICS filing system: a new and very different storage system has been incorporated. The backup system has also been changed to deal with the new storage system. The concept of logical volumes has been introduced. A logical volume encompasses a set of real devices. The directory hierarchy is on a separate volume, and any directory in the hierarchy can be placed on a new volume, with all its decendants on the same volume. For dumping onto tape a volume dumper is used which dumps one volume at a time; no tree walk is used. Each physical device has a contents table which is a list of pages of the segments on the device. Incremental dumping is done for pages on volumes rather than files. Thus, the scheme used is basically the same as the one proposed by Stern, but used at a lower level (for volumes and pages rather than for the hierarchy of files).

The EMAS system [EMAS74] provides an automatic checkpointing facility for files. Files are part of the user's virtual memory and cannot be accessed through the paging mechanisms until they have been connected (i.e., the virtual memory disk address mapping has been set up). When a user process is created, its virtual memory space is created, initialized, and copied to disk. When the process is run, the working set is in main memory, and pages are transferred back and forth between core and drum. A page may be forced to disk because the drum gets full, or the process becomes dormant again. If a page is forced to disk, all of the updated virtual memory pages are forced to disk at the same time. This mechanism is required by the "consistency" rule in the EMAS system. Therefore, a suitable restart copy of the virtual memory of a process (which includes the files) is provided on the disk. The problem of inconsistencies between the state of the process and the states of its associated files is avoided because the filing system uses the resources provided by the paging system. The paging system assumes complete responsibility for maintaining a consistent backup copy of all the state variables of the process (including files). Consequently, if the consistency rule is always obeyed, automatic checkpointing is provided.

Incremental dumping can be done as part of an audit trail scheme [MASC71, RAND 70, MASC73]. An audit trail gives only the changes made to files from given states onwards. These states are redefined regularly for reasons of efficiency (so that audit trail journals do not become too long). For example, in a system described by Wimbrow [WIMB71], files are dumped when they have to be reorganized because they have become disorderly as a result of the operations performed. In the CMIC system [GIOR76] all files are checkpointed regularly at moments when no user has the database open.

Another scheme designed for System R [LORI77], but not implemented, works as follows (see Figure 3): Each segment (which is similar to the concept of file) consists of a page table with pointers to the data pages. Associated with each pointer in the page table are three bits: a shadow bit, a cumulative shadow bit, and a long term shadow bit. When a segment is updated a backup and a current copy are maintained (in a way to be described later). For every page which is updated, the shadow bit and cumulative bit are set in the page table entry of the segment containing the page. Thus, for example, in Figure 3, pages 13, 6, and 21 are updated and therefore replaced (for reasons described later), in this case by pages 5, 3, and 19. The shadow bits and cumulative shadow bits for those pages are, therefore, set in the current page table. The cumulative shadow bits for pages 8, 21 (now replaced by 19), 10, and 4 were set already.

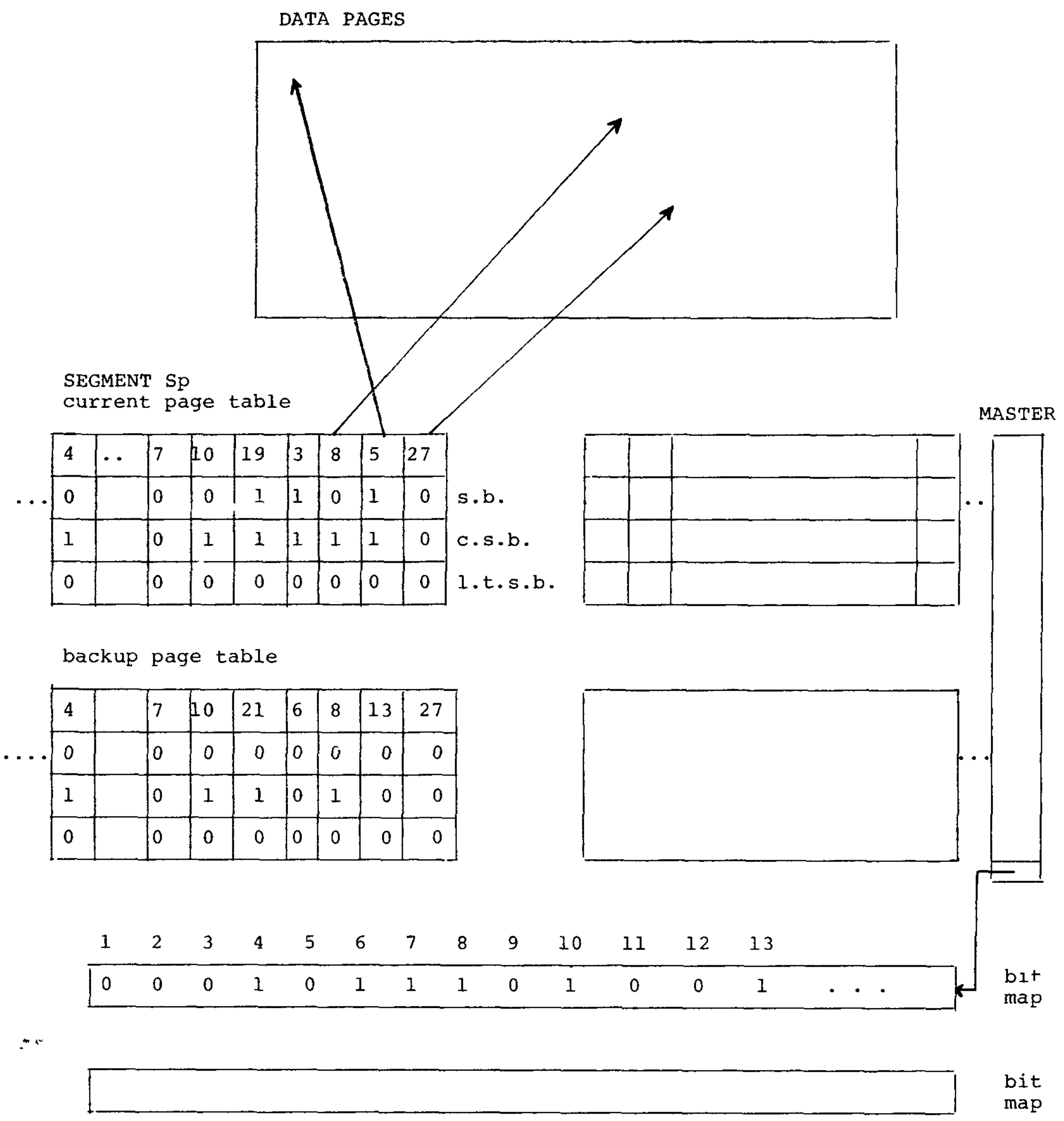
When the current state of the segment is saved (i.e., replaces the old copy), the shadow bits are switched off, and the old pages of the backup version, having been replaced by the new versions from the current copy, are released. Checkpoints of all the segments are taken regularly. This involves the copying of all the page tables for which at least one cumulative shadow bit is switched on; the cumulative bits are copied into the long term bits and then switched off. A process P is started periodically. Process P is a system process which copies onto tape all of the pages of all segments in the systems for which the cumulative shadow bit is on at checkpoint time. The long term checkpoint bits are used to make sure that subsequent saves will not release the pages before P has copied them.

If in Figure 3 the transaction were closed in the situation shown, the contents of the current page table would be copied into the backup page table. If, subsequently, a checkpoint of the segment page table were taken, the cumulative shadow bits would be copied into the long term shadow bits, involving the setting of the long term shadow bits for pages 4, 10, 19, 3, 8, and 5. All of the cumulative shadow bits would then be switched off.

A special checkpoint file is used in HIVE [TAYL76] to record transactions. Information put in the checkpoint files can be recovered after a crash to restart those transactions. Individual transactions can also be reprocessed using this file.

AUDIT TRAIL

An audit trail records the sequence of actions performed on a file [BJOR75]. The audit trail contains information about the effects of the operations, the times and dates at which the operations occurred, and the identification codes of the user (or user programs) issuing the operation.



4	..	7	10	19	3	8	5	27	
0		0	0	1	1	0	1	0	s.b.
1		0	1	1	1	1	1	0	c.s.b.
0		0	0	0	0	0	0	0	l.t.s.b.

4		7	10	21	6	8	13	27
0		0	0	0	0	0	0	0
1		0	1	1	0	1	0	0
0		0	0	0	0	0	0	0

1	2	3	4	5	6	7	8	9	10	11	12	13	
0	0	0	1	0	1	1	1	0	1	0	0	1	. . .

FIGURE 3. The implementation of segments in System R.

Audit trails can be used for several different purposes, such as:

- *Crash recovery.* If backup versions of files are reinstalled, an audit trail can be used to perform operations on them, thereby restoring their states at the time of the crash [CURT77].
- *Backing out.* If a system crashes (without damaging secondary storage) the files affected by the processes running during the failure can be restored to their states before those processes started. The audit trail can be processed backwards for backing out. Also, a single transaction (or job) can be backed out in case a deadlock occurs or the transaction fails. The data affected by the transaction can be restored to their state before the transaction (or job) was started.
- *Certifying system integrity.* The audit trail can verify that rules and policies dictated by laws, business agreements, and the like, are being followed [BJOR

75]. Bjork has concluded that audit trails will be the major integrity tools for shared data usage beginning in the late 1970s. However, there are reasons to believe that other techniques will also be used. For example, the techniques of differential files and incremental dumping could provide the same facilities, although in a more complicated way.

A very extensive description of the use of audit trails for crash recovery and backing out has been given by Gray [GRAY77].

Traditional recovery techniques for filing systems [FRAS69, WILK75] may be insufficient to prevent the loss of the changes caused by the most recent operations performed in the filing system. The usual method, incremental dumping, checkpoints the files at regular intervals, but operations performed on files after the last checkpoint will be lost if a crash occurs. This does not matter in many operating systems because jobs can be resubmitted or operations can be redone. However, in systems where updates are made online from different sources, such as in banking or airline reservations [MASC71, RAND70, WIMB71, TONI75], this method may be unacceptable: one cannot afford to lose any update should such systems fail.

Another reason that traditional recovery techniques for filing systems may be insufficient is that data management systems are physically organized very differently than filing systems (some implementations of relational database systems might be said to be exceptions to this general rule). The difference in design parameters would make a scheme such as Fraser's unsuitable for most data structures used in existing database management systems. In systems like these an audit trail can provide a solution. Before a transaction is performed in the database, it is recorded on the auditing tape; this procedure is carried out without buffers [WIMB71] or by implementing a "log tape write ahead protocol" [GRAY77] to protect against crashes that destroy the mainstore contents.

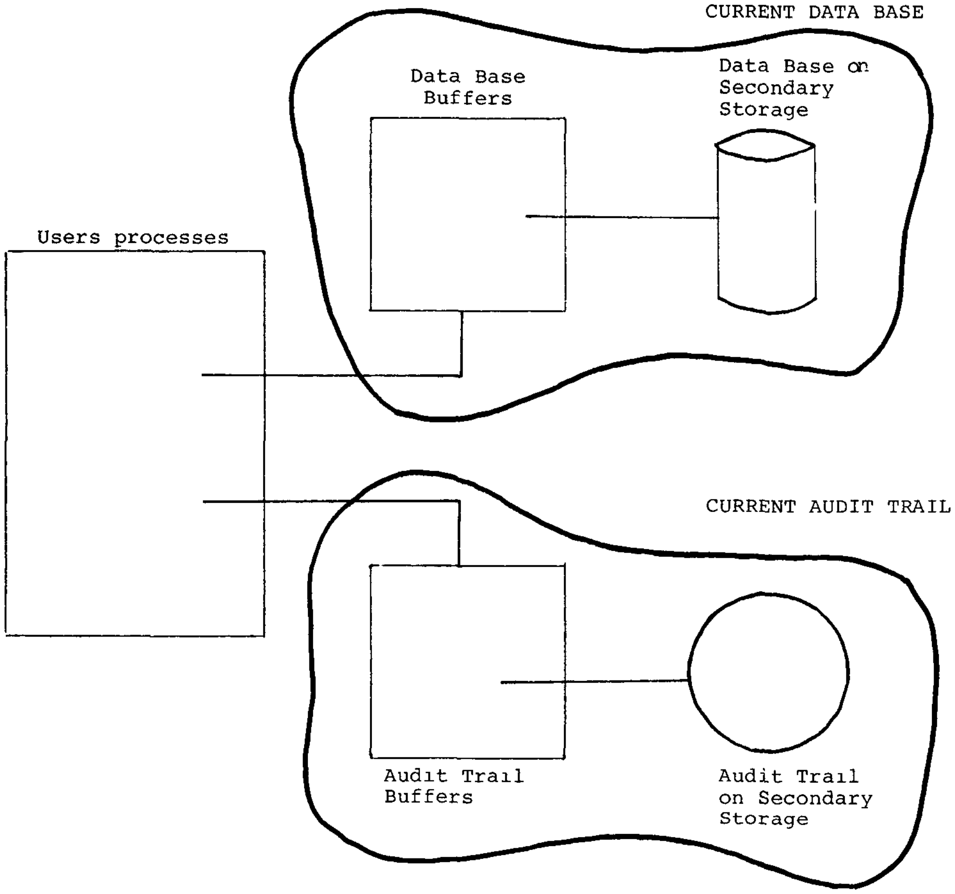
Buffers (see Figure 4) may lead to inconsistencies between the database and the audit trail [GIOR76]. If the buffers are lost after a crash the database will, in general, be in an inconsistent state, and the audit trail will be incomplete; so it cannot be used to restore the correct state (as illustrated in Figure 4). (However, another possibility would be to salvage the buffers from main storage after a system crash, thus making possible the proper closing of the audit trail tape. This has been tried, not always with success, in IMS [IBM] systems using core memory and in the CMIC system [GIOR 76]. The buffers cannot be salvaged if the contents of main storage are lost after the crash.) The log tape write ahead protocol avoids this problem.

The audit trail can be used to back out a process which may have interacted with a second process in such a way that the second process will have to be backed out. The audit trail can then be used to back out that second process which may have interacted with a third process, and so on. Thus, using an audit trail to back out unfinished transactions performed by interacting processes, can lead to a "domino" effect [RAND 75, GIOR76, CURT77].

A locking scheme, such as the one used in system R [ASTR76] and many commercially available systems [CURT77], can be used to avoid these problems by making these interactions impossible. This solution can only be applied if these interactions are not necessary (i.e., they are accidental rather than deliberate). The backing scheme will make it possible to use an audit trail (or a "log" as it is called in many systems) to undo partially finished transactions.

Special checkpoints made at moments when no user is active [GIOR76], have been proposed, since it is not possible to back up past such checkpoints, and thus the domino effect would be halted at these points. However, such occasions occur too infrequently in busy systems for this scheme to be helpful; frequent forcing of all users to become inactive would be impractical.

Another form of audit trails appears in a system described by Lampson and Sturgis [LAMP76], under the name of *intention lists*. An intention list specifies the operations to be performed by a processor. A processor, which is a node in a network,



The current data base is always consistent with the current audit trail, however the data base on secondary storage is, in general, not consistent with the audit trail on secondary storage.

FIGURE 4. A general database system using audit trail.

may receive an intention list, containing the specifications of the operations to be performed on its local database. Intention lists, like the audit trail used in the CMIC system [GIOR76], can be reprocessed without backing out the interrupted process. Intention lists, once received and accepted, cannot be lost unless a catastrophe occurs, such as a head crash. They are safeguarded because they are stored on disk at a fixed place and are not altered when processed. Unless the processor malfunctions, the operations specified in the intention list will always be done.

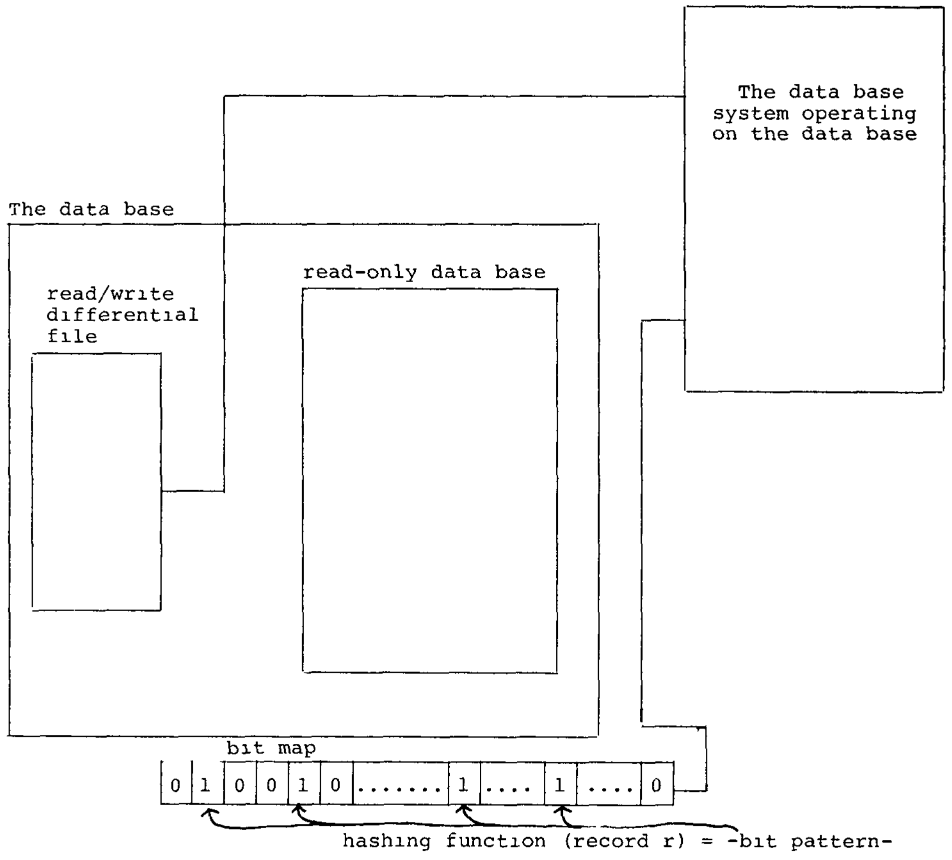
Whereas audit trails record completed database updates, intention lists contain operations not yet performed (although physically the audit trail may be written first, as is the case when log tape write ahead is used). In other words, issuing an intention list is an event that could be registered in the audit trail. Processing the intention list is similar to processing an audit trail during crash recovery or backing out.

DIFFERENTIAL FILES

Under the differential file (also called "change set") scheme the main files are kept unchanged until reorganization. All changes that would be made to a main file as a result of transactions performed are registered in a differential file. The differential file will always be searched first when data is to be retrieved. Data not found in the differential file is retrieved from the main database. The most recent entry for a given record in the differential file must always be retrieved.

Severance and Lohman [SEVE76] fully describe the technique, and also an efficient hashing method to implement it (see also Figure 5). They use a small associative memory, in the form of a bit map accessed by the hashing scheme, to reduce the probability of making an unnecessary search in the differential file. The database system checks the bit map (see Figure 5) to see if the bits for a record are set or not before accessing that record. If the bits are set the record is probably in the differential file. It is shown analytically how to keep the probability of a filtering error low; this error occurs only when the bit map suggests wrongly that a record is in the differential file. The hashing function maps the record address onto a number of bits in the bit map (see Figure 5). The bits for a particular record may be set because each of them occurs in at least one set of bits associated with another record in the differential file.

Severance and Lohman [SEVE76] also describe the advantages of differential files for recovery, integrity, the implementation of incremental dumping schemes, and the simplicity of software. Another advantage claimed is the possibility of performing


The data base system operating on the data base
The data base
read/write differential file
read-only data base
bit map
0 1 0 0 1 0 .... 1 .... 1 0
hashing function (record r) = -bit pattern-

The bit map suggests that record r is in the differential file, because the bits set in the bit patterns produced by the hashing function are set in the bit map.

FIGURE 5. A differential file technique using a hashing scheme.

queries which do not need the exact values of all files; such queries access a suitable (current) view of the data without locking out the update transactions.

The disadvantages of the approach using differential files are [LORI77]:

- An access to a data element appears slow: if an initial search of the differential file reveals that the data element is not among the modifications, the required element must be fetched from the database. However, Severance and Lohman show that this problem can be almost completely overcome with hashing for many systems and applications; they also show how to construct a good hashing scheme for particular systems.
- Eventually the differential file must be merged with the main database—an operation which can be slow. This will certainly be a big problem if the system needs to be available without interruption.
- Since an update can affect an element which has already been modified, the differential files must be suitably organized. A hashing scheme [SEVE76, RAPP75] ameliorates this problem.

Differential files are, for example, used in the VADIS system, where they are called MODFILEs [RAPP75]. For every file in the system there is a MODFILE. The system has been developed to facilitate recovery after power failure. Completed transactions will never have to be undone after a power failure. Transactions not completed before the failure are not undone explicitly; but their effects are ignored using the MODFILEs and a TRNSDONE file as follows:

1) Each entry in a MODFILE has a header with: record type, transaction code, pointer to previous modification of the same record, time, transaction number and some other identification codes.
2) There is a TRNSDONE file which contains the numbers of the completed transactions.
3) For every record fetched from a MODFILE the transaction number is compared with the TRNSDONE numbers and the current transaction number.
4) If the number is neither in TRNSDONE nor is the number of the current transaction, then the previous version of the record is taken (the one pointed to by the retrieved entry from the MODFILE, or in the main file), because this means that the record was put in the MODFILE by an uncompleted transaction before a failure.
5) MODFILE entries and system variables are forced out to disk at the end of each transaction. Thus the entire MODFILE mechanism is checkpointed after each completed transaction.

Differential files are used in a system, described by Titman [TITM74], for both efficiency and reliability. The way in which ordinary files are kept makes insertions or deletions very expensive. In Titman's system the files are binary relations which are stored in highly compressed fixed-length blocks. Elements are identified by a block number and the sequence number of the element in the block. An insertion or deletion requires the complete reorganization of the file giving the elements new identifiers. For reasons of efficiency, an "add set" and a "delete set" of elements are kept for each file. For reasons of reliability, a "change set" is also kept for each file; it is used to register the changed records. The add set, delete set, and change set together form the implementation of a differential file. The main files are kept on a separate device which is never written on except during reorganization. These files can be duplicated on tapes for recovery. Checkpointing is carried out by saving the add, delete, and change sets.

BACKUP AND CURRENT VERSIONS

Backup versions of files or databases can be kept in order to make possible the restoration of the files to a previous state.

For example, many file-editors produce a complete new version of a file while a user is editing a file. The original file remains unchanged during the edit-session. The new version is a complete new copy; it is not achieved, for example, by using a differential file. If a user notices a blunder during the edit session, the original copy of the file will not be lost.

Incremental dumping (of current versions) can be used as a utility to maintain a backup version of a filing system or database. Copies of altered files can be used to update a backup version of the whole system or database. This is done in the Cambridge filing system [FRAS69], where two processes are used: one makes incremental dumps and the other creates versions of the system.

Similarly, complete copies of the database can be made regularly in order to make possible the restoration of the database to an earlier state. For example, the original version of MULTICS [DALE65] prepared a weekly dump; it included all files which had been used within the last several weeks plus all the system files.

An optimized version of the backup/current versions has been designed for System R [LORI77] and is used in other systems for segments (synonymous to the notion of files). Figure 3 illustrates this mechanism. (Figure 3 is not complete; for example, the MASTER is duplicated. However, sufficient details are shown to illustrate the mechanism discussed here.) For each segment a page table is used to locate the data pages. There are two copies of each page table, which are identical after a SAVE.SEGMENT. If a page of a segment is altered for the first time after a SAVE.SEGMENT or OPEN.SEGMENT operation, its new value is put in a newly allocated page, and the current version of the page table is updated to point to the new page.

For example, in Figure 3, page 13 in segment Sp is updated. The new value of the page is therefore, in this case, placed in page 5, and the current page table is made to point to page 5 instead of page 13. The backup version remains unaltered. When a SAVE.SEGMENT is issued the buffers are forced to disk, modified pages are released (i.e. the old versions), and the current version is copied into the backup version. So, for example in Figure 3, pages 13, 6, and 21 will be released.

This releasing of pages causes the bit map used to indicate the free pages to be updated. In Figure 3, the bit map will have to be updated so that bits 13, 6, and 21 are reset, and 5, 3 and 19, are set. Two copies of the bit map are maintained. A MASTER table points to the current map. The current bit map always reflects a checkpoint state of the system (i.e., all of the pages pointed to by the backup versions of the page tables). The SAVE.SEGMENT operations can be done at random moments in time.

This current and backup versions recovery technique is used in System R for recovery from system failures to restore a checkpoint state. The log (audit trail) is then used to redo transactions completed before the failure. In addition the log is used to back out transactions which were incomplete at the checkpoint time. The log is also used in System R to back out transactions when a transaction fails or a deadlock occurs.

The backup/current version technique described above, is not designed in System R for recovery after a failure where secondary storage is destroyed; incremental dumping is used for this purpose instead. After the SAVE.SEGMENT operation the MASTER table is made to point to the up-to-date bit map. This scheme provides the possibility of restoring a segment to its last consistent state (held in the backup version) and of restoring consistency after a system failure. The operations of unfinished transactions, performed before the failure will be lost; these transactions will have to be restarted.

Physically separate backup and current versions of the page table provide backup and current versions of the segment under this scheme. The logically different versions of a segment overlap (physically) in their implementation where they are the same.

MULTIPLE COPIES

The technique of multiple copies includes two techniques:

- Keeping an odd number of copies of the data. If a majority of the copies have the same value, then that value is taken as the correct one. This technique is then called *majority voting.*
- Holding two copies with flags to indicate "update-in-progress." An incon-

sistent copy (or suspicious copy) is always recognizably inconsistent, because of the flags used; if the system crashes during update the flag will still be set after the crash.

Except during update, the multiple copies must always have the same value. If the different copies are updated by the same processor then an "update-in-progress" flag (or "damage flag" [CURT77]), used if there are only two copies, provides crash resistance. A consistent copy can always be retrieved after a system restart; this copy will have either the value it had before the update in progress during the crash, or the new value. The inconsistent copy can always be recognized as such and discarded. Majority voting may also be used to detect incorrectly performed operations; this is especially useful if different processors update the different copies. Faulty processors can then be detected and ignored or disconnected.

The important difference between the multiple-copies technique and other techniques, such as backup/current version or careful replacement, is that with the multiple-copies technique the different copies always have the same value except during update of the actual physical data structure. Thus, if the two copies hold consistent values they must be equal. Backup or incremental dumping schemes could also be used to keep different copies that hold the same value except during a job or transaction. In these cases there is always a primary copy and a backup copy, which may hold different, but consistent, values from the database point of view. The technique of copying onto tape at the completion of each updating, is different from the multiple copies technique. The multiple copies must exist all the time; physically, they do not overlap, and they have equal status. Schemes based on different backup or archival versions, are definitely not examples of the multiple copies technique.

Majority voting data has been used extensively for space flight applications, such as in the space shuttle system where four computers are configured to receive the same input data and calculate the same outputs [SKLA76].

The technique of two copies with flags is used in recovery for segments in System R [LORI77]. A MASTER table is used to indicate which segments are open or closed and which bit map (two copies are held) is up-to-date (see Figure 3). Two copies of the MASTER table, both containing the same information, are kept to ensure that, if the system crashes while the MASTER table is being updated, only one copy will be left behind in an inconsistent state; the other copy has either the new state or the state the table was in before the update started. (Only one copy of the MASTER table is drawn in Figure 3, but two copies are used as described above.) The copy that is in an inconsistent state can always be identified.

Similarly, two copies of the MASTER directory in the filing system of GEORGE 3 are maintained [NEWE72], and two bits to make possible the distinction between a valid and invalid copy after a crash during update.

System HIVE [TAYL76] maintains two read-write versions for every file. This provides one of the characteristics of a cycle (see the section on salvation programs): The local effects can be undone as long as the cycle has not yet finished. During a cycle one of the two versions is updated. At the end of the cycle this version is copied onto the other version. The system knows which of the two versions is the one updated during a cycle. Crash resistance is therefore provided for individual files. Apart from this, a checksum is maintained for each version. This, generally, enables partially updated or corrupted files to be detected. In general two copies and two flags (bits) are sufficient to provide crash resistence. The flag indicates whether a copy is suspicious or not. If the two copies are updated immediately after each other, as in System R, then a copy is likely to be inconsistent if its flag is set. In system HIVE, however, the two copies are kept on separate storage devices, so the checksum allows detection of incorrectly performed operations. Although selective redundancy is not uncommon, system HIVE is one of the few existing systems in which more than one complete copy for every safeguard file is maintained to provide crash resistance. One copy is updated during a cycle, and the second copy is updated in pages which correspond to the changed pages of the first copy.

CAREFUL REPLACEMENT

The objective of careful replacement is to avoid updating data structures "in place." (Some bit or pointer must always be updated "in place" [NEWE72, VERH77a]; however, this feature will not be elaborated further in describing the careful replacement technique.) The update is performed on a copy of a component (record, page, disk-block), which replaces the original only if the update is successful; and the copy is kept until after the replacement is made successfully. The same is done for objects in the data structure which point to those objects. There are two instances of the data structure only *during* update; otherwise there is just one copy, which contains the current value.

This technique differs from differential files, which accumulate update requests for unaltered originals. However, careful replacement could be used to merge the main file with the differential file. With the careful replacement technique the two "virtual" copies are held only during an update (or perhaps within a recovery scope specified by the programmer [VERH77a]); this makes the update or sequences of updates as safe as possible by reducing the chance of being left with an inconsistent copy or mutually inconsistent files. The two instances of the data structure are referred to as two "virtual copies" because they overlap in identical objects. There are two different descriptors to access the two different instances.

This technique is fully explained by Gamble [GAMB73] for a filing system. Files consist of data pages pointed to by a tree of directory pages. A master directory points to each top directory page of the files. If a file is updated using careful replacement, it can always be restored to its state prior to update.

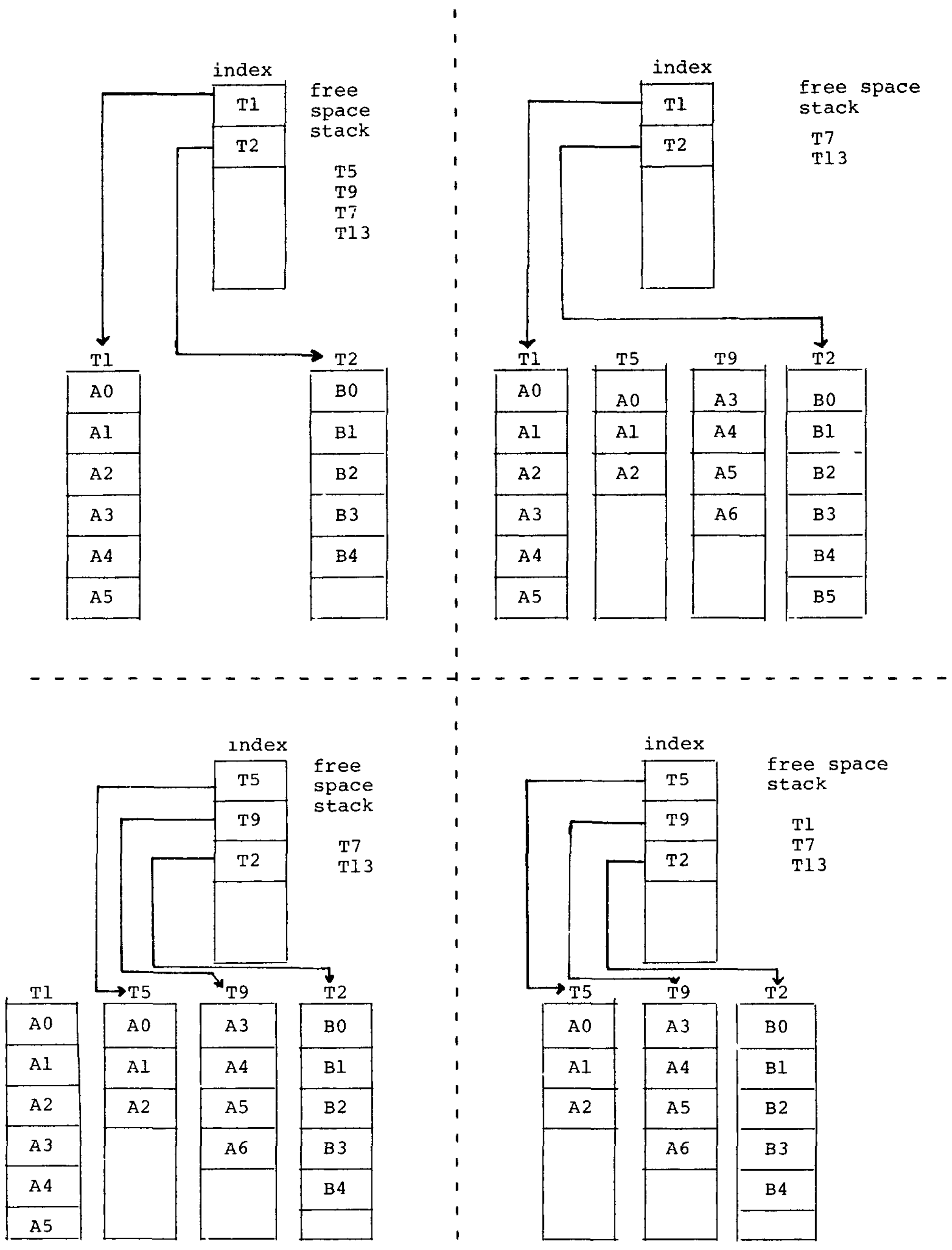
This approach avoids the three disadvantages of differential files [LORI77] mentioned above. However, careful replacement has its own disadvantages:

- The file or data structure must be tractable. For example, implementing a file as a list of linked pages could be prohibitively expensive. This expense arises because replacing a page requires updating the link in the page pointing to it. Careful replacement requires updating that link in a copy of the page. Thus, the change in one page can propagate replacements through the whole list. The constraints and requirements that careful replacement sets for the data structures are fully described elsewhere [VERH77a, VERH77b].
- Overhead costs are incurred in disk accesses. In GEORGE 3, where this technique is used for files, but not for the much more heavily used MASTER-directory, the measured overhead reported by Newell is surprisingly low. The method is also used in the MU5 system [GAMB73]. So the overhead can be a disadvantage which does not outweigh the advantages.

Files in the system described by Lampson and Sturgis [LAMP76] are updated using careful replacement during the processing of the intention lists. System R uses the basic idea to update segments.

The CMIC system also uses this technique [GIOR76]. The storage structure used is similar to B-trees [KNUT73]. If an insertion is made in a full track, two new tracks are obtained and the contents of the full track plus the new entry are put in these two new tracks (see Figure 6). The same is done for the index table which contains the pointers to the data tracks.

This method also uses the *leaf-first rule*, which states that copying of information to slower memory (e.g., from main storage to drum, or from drum to disk) is done in such a way that no descriptor or pointer can ever reference a block at a faster level of the device hierarchy. This ensures that the (sub-)structure on mass storage is always valid, for it will always be a valid and consistent tree. Every pointer or descriptor on mass storage always points to valid structures on that mass storage: no parts of that data structure will still be on faster mem-

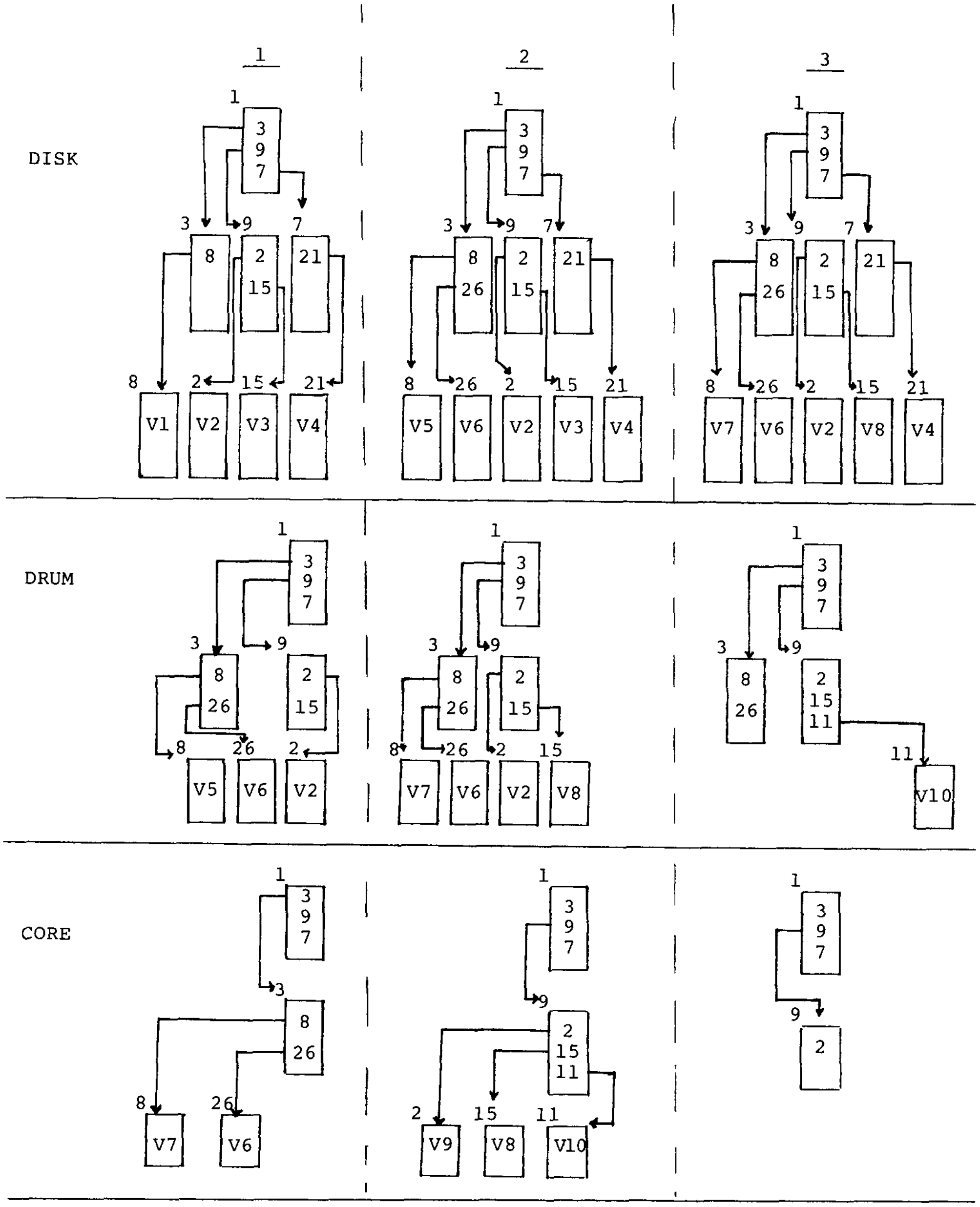


The free space stack contains the addresses of the free records.

FIGURE 6. The insertion of an entry A5 in a full track in a storage structure as used in CMIC, in four steps

ory. It also ensures that no data on mass storage is ever removed from the structure during update. Instead, replacement is used by using new tracks when necessary. Data structures, pointers, and descriptors in the tree structure on mass storage are always present; they are not removed and put back during updating.

The careful replacement technique is often used in filing systems using a hierarchy of devices by employing the leaf-first rule and the *root-segment rule* (see Figure 7).



Three possible subsequent situations.
Where: Vi = a value of a data page

FIGURE 7. The careful replacement technique in a hierarchy of devices.

The root-segment rule states that if a data page is on a particular level of storage in the hierarchy, every directory page between it and the root of the file is on that level or a faster level (the root is the top directory in the tree of directory pages). These two rules mean that careful replacement is used at every level in the hierarchy. Were the contents of the main memory to be lost after a crash, the drum and disk would still have two copies of the file: the disk copy contains an old value, the drum copy contains a newer value of the file (some pages of the file are on drum and others on disk).

Figure 7 shows how updates on a main-memory version of a file are subsequently made to the copies of that file held on drum and disk using these two rules. The example in Figure 7 shows the pages of a file as held on disk, drum, and in core during processing, at three different subsequent moments in time: time 1, 2, and 3. At time 1, the copy on disk shows that the change made to page 8 and the addition of page 26 into directory page 3 have not yet been consolidated in the disk copy of the file. The copy in core shows that page 8 has been updated again, and that the update has not yet been consolidated on the drum. Similarly, at time 2 the copies on disk, drum, and core show that the update of page 8 and the addition of page 26, consolidated on drum at time 1, are now consolidated on disk. The update of page 8 in core at time 1 is now consolidated on drum. In the meantime other updates and additions have been made to the file in core. Finally, the copies at time 3 show yet more updates, and show how they are consolidated.

Although pages are updated "in place," this technique can be classified as being a careful replacement technique because updates are made in fast memory, and the replacement takes place on slow memory containing the files. It is a careful replacement technique because every valid tree structure is consistent. If the contents of both main memory and drum are lost, there will remain a consistent copy on disk. If the contents of main memory are lost, the updates which were started on the main memory copy but not consolidated in the drum copy yet, will be lost. If the contents of the drum are also lost, the updates which were consolidated in the drum copy, but not yet in the disk copy, will be lost as well.

A system using the leaf-first and root-segment rules has been described by Schwartz [SCHW73]. Files are trees of pages which are either index pages (i.e., directory pages) or data pages (as in Figure 7). A file descriptor is the root of the index table; a directory file contains the descriptors. In that system, and in CMIC too, the directory file is updated "in place." If absolute crash resistance were to be provided in this system, the multiple copies technique could be used for the directory, as is done in GEORGE 3 [NEWE72].

If the recovery is to be provided for transactions consisting of more than one update, replaced pages can be updated "in place." This technique has been used at Newcastle [VERH77a] to provide recovery for files within user defined scopes. Scopes can be nested, as with "spheres of control" [BJOR72]. These scopes are termed "recovery blocks" [RAND75]. A recovery block consists of an acceptance test and a set of alternative algorithms. The first algorithm is executed first, followed by the evaluation of the acceptance test. If an error occurs during the execution of an alternative, or the acceptance test is not done successfully (either it fails or it is never done), all the operations performed so far are undone, and the next alternative is invoked, if possible. (A power failure still causes all the operations to be undone [VERH77a] however.)

The same is now done for the second alternative. If all the alternatives of a recovery block are exhausted, an error is generated in the outer recovery block, or the program fails if there is no outer recovery block. A copy of a file is maintained for every level of nesting in which the file has been operated upon; these copies contain identical nonupdated pages. The current value of the file is in the latest copy. Recovery means restoring the copy as it was before the current recovery block was entered. On exiting a recovery block successfully, the updated part is substituted in the original that existed just before the recovery block was entered.

DIRECTIONS FOR FUTURE RESEARCH AND RELATED WORK

Recovery among interacting processes has not been dealt with explicitly in this survey. These problems can be significant. For example, the designers of System R encountered major difficulties when trying to use the shadow mechanism (as in the backup/current version technique described previously) for recovery in a multiuser environment. The scheme is now used only for system recovery after a total system failure. Some of the problems of providing recovery for interacting processes have been described elsewhere [RAND75, GIOR76, ASTR76, RAND78, CURT77]. This topic is a subject of ongoing research. A major difficulty is that processes being backed out past interactions performed may require other processes to be backed out, creating a "domino" effect [RAND75].

The problems with the domino effect could be explained by generalizing Russell's diagrams [RUSS77] for producer-consumer systems. Vertical lines, in such diagrams are then used to denote the progress made by processes in time. An arrow from one process A to another process B occurs in such diagrams if A last updated data that is read by B. (See Figure 8.) This arrow now denotes that effectively a message was sent by A when it updated the data and was received by B when it read that data. Each time the data is read a message has been sent and received.

As shown by Randell [RAND75], it is not known which processes to checkpoint at which moments in time unless this diagram is known in advance. Russell [RUSS77] makes use of that knowledge for his producer and consumer systems. The diagram is not known in advance in a database system, since it is not known which operations (programs), sharing the same database, will be performed in parallel. Even if it were possible to predict which programs will run concurrently, it might not be possible to predict which interactions will occur.

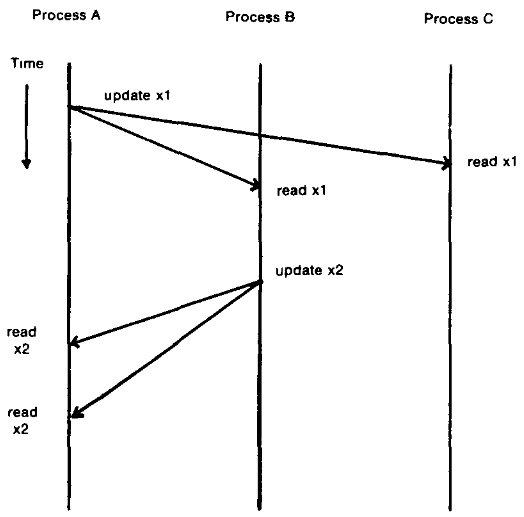


FIGURE 8. A generalization of Russell's diagrams [RUSS77].

It seems that in our diagram (Figure 8) a process could be checkpointed each time it has updated some global data, and before some other process can read that data. This could be done using techniques similar to cacheing as discussed by Randell [RAND75]. However, the system still has to keep track of the interactions that have taken place (i.e., see Figure 8) to determine which other processes to back out to which specific points in time.

The cacheing technique could be used to reset a process; a recovery technique can be used to back out the database; and the same or some "companion" technique could be used for each process "to put messages back." This latter technique would have to provide the facility to reread the same value without restoring the shared data, since that may have undesired effects on other processes.

The main problems are to keep track of the diagram, and given the diagram, to decide upon which processes to back out to which points in the event of a process failure. These problems are at present being addressed at the University of Newcastle upon Tyne.

There are two approaches to the problem at present:

- *Prevent the interactions.* Preventing interactions is feasible only when the interactions are not required. In this case processes can be executed one at a time, in some sequence which can be implemented by explicit locking [GRAY76] or implicit locking [BANA 77]. The explicit locking schemes are widely used in database systems, most of which abandoned implicit locking a few years ago.

A user or a program can request various modes of access to an object; these modes include exclusive or shared access. Some access modes are incompatible: if one user has access to an object in a certain mode, then another user is refused access to the same object in an incompatible mode. For example, multiple reads of a file are allowed, but writing precludes other file operations. Such locking schemes prevent unwanted interactions, while allowing shared access when it will produce no interference among programs.

Implicit locking occurs in monitors [HOAR74], which are used to implement resource allocation algorithms. Programs seeking to acquire or release resources invoke monitor procedure calls. The monitor locking scheme ensures that only one process at a time will update an object; hence uniprocess recovery schemes are possible. Other processes are prevented from updating that object until the process which is updating that object concludes a unit of recovery, such as a transaction [GRAY77] or a recovery block [BANA77]. However, locking by using monitors (or "serially reusable programs" as they are called in OS/360) has been shown to be disastrous in database systems, for high concurrency.

A similar way in which the restrictions can be enforced is by using a capability architecture [GRAY70, DENN76] to implement a high degree of error confinement [LIND76]. Capabilities permit sharing, but will, when used wisely, limit the number of processes to be backed out. A capability is a protected key or password for using storage objects and procedures. Capabilities can, therefore, in fact be a means of implementing locking. The use of capabilities can prevent unwanted interactions like locking schemes, and therefore limit the risk that errors will lead to much damage before being detected.

- *Synchronize the processes with respect to recovery.* To avoid arbitrary backing out among processes that must interact, severe constraints are needed. Randell suggests a "conversation" [RAND75] which is a recovery block with locks that make all processes enter and exit together. Processes in a conversation may not interact with those outside. In case of failure, processes need to be backed out only to the beginning of their conversation. "Recovery lines" are needed to back out all the processes [RAND78]. A recovery line is a set of consistent checkpoints.

These approaches are designed to overcome the domino effect of backing out interacting processes. At worst, all the processes that have interacted with a failing process will have to be backed out to their initial states. The two approaches above force the creation of recovery points (i.e., checkpoints) in such a way that recovery lines are formed to minimize backing out in case a failure occurs.

The first approach is well understood, and many systems use successful schemes based on it. One efficient way to implement the first approach is used in IMS. The system queues all modified records, between checkpoints; this is similar to constructing an intention list [LAMP76]. Thus, if a program updates records, other programs are prevented from accessing these updated records until that program terminates or issues a checkpoint. The program can be backed out to its last checkpoint, without undoing effects to other programs, before it terminates.

The second approach, however, is not yet well understood. If the first approach has not been used, it is generally, in existing systems [CURT77], up to the operator or data administrator to analyze the situation and take appropriate action. Whenever the first solution is not possible, many systems accept the possibility of the domino effect. These systems must fall back on checkpointing the total system at regular intervals [GIOR76], or synchronizing the checkpoints of the various programs so that rollback can be performed to a common point in time [CURT77]. This solution may be too cumbersome to be acceptable in many distributed systems.

This is one of the major problems of integrity in distributed database systems. Also, the implementation of the first approach (i.e., prevent the interactions) in distributed systems, such as systems using intelligent terminals for example, is not trivial. Gray [GRAY77] describes a "recovery protocol" which makes recovery possible by implementing two important requirements: the decision to commit the operations performed during the transaction on data at different nodes in the distributed data base system (i.e., the successful completion of the transaction), has been centralized in a single place (i.e., the decision is taken by one processor).

This survey has discussed recovery by state restoration. Whenever a failure occurs, a state which is believed to be error-free is restored before attempting to continue further operation. Other recovery techniques are still under investigation, for example:

- *Error diagnosis and repair.* Instead of restoring a state when an error is detected, an attempt could be made to identify and repair the cause of the error [RAND78]. This may be very difficult, not only because different errors may be caused by one fault, but different faults may cause the same error.
- *Compensation.* Rather than undoing operations by state restoration, it could be attempted to nullify the impacts of these operations by compensating for their effects. This can be achieved by providing supplementary corrective information [DAVI72, RAND77]. For example, if a database had been updated to indicate that an employee has been given a \$1000 wage increase, instead of an intended \$100 increase, then a wage decrease of \$900 could be given as compensation. This could be cheaper than backing out the transaction on the database.

Little has been said here about the costs of recovery itself. A recovery procedure may put the data structures back into consistent states; or it may restore the data to a previously existing state. There are few papers which have examined the most common failures or the degrees of recovery suitable for different environments. This survey indicates guidelines rather than detailed analyses of some degrees of recovery.

The cost of the error detection scheme must also be taken into account. There have been few papers reporting systematic approaches to software error detection. Yet, error detection is absolutely essential to make recovery techniques useful. Many system structures, system concepts, and language concepts have been developed to make the inclusion of tests for error detection easier. They provide a framework which allows the user or programmer to embed such tests in a structured manner. Examples of this are capability systems [DENN76], recovery blocks [RAND75], and security and access control as provided in recently designed languages. However, these concepts only provide a framework in which to embed tests, rather than the actual error detection mechanisms themselves. Error detection schemes that employ tests could be designed using two approaches: 1) Test if algorithms perform completely according to their specifications; and 2) Distinguish certain types of errors, and test for their presence (or absence).

Testing the validity of all of the input data and parameters of procedures is an example of a systematic error detection scheme based on the second approach. However, few systematic ways in which tests can be constructed, using either approach, are reported in literature [RAND78]; little is known about the costs of error detection schemes. Error detection in system software is generally done in an ad hoc fashion using the second approach. Some initial work on the construction of (run time) tests, using the first approach, has been described elsewhere [VERH77b].

The techniques described in this survey provide recovery for files based on secondary storage. Verhofstad has some preliminary extensions to data types more complex than files, and recovery procedures for different levels of a system [VERH77b]. Randell has discussed "nested recovery" using recovery blocks [RAND75]. The relation to careful replacement has been discussed by Verhofstad [VERH77b]. Beyond such preliminary works, little is published about the systematic recovery of program or data objects.

SUMMARY AND CONCLUSIONS

This paper has described many of the techniques used to implement backing out, crash recovery, crash resistance, and consistency. These techniques can be used in different environments, for different purposes; they can complement each other. Figure 9 is a cross-reference table; it shows which techniques discussed in this survey are used in which particular systems. This table may be incomplete for the systems it covers (e.g., System R may have some sort of salvation program); however it summarizes the most important features of the systems as reported in the literature.

It appears that for filing systems, where short term losses are not considered serious, the combination of incremental dumping, a complete backup version of the system, and a salvation program suffices for a high degree of reliability. This approach has been successful in MULTICS, the Cambridge system, and EMAS. A salvation program may be needed to be sure data is consistent after a crash; its use may cause some data to be lost.

This combination can be improved by an audit trail, a safeguard against losing updates; this is done in IMS. The recovery facilities in IMS are extensive but not systematic; there is neither a general approach nor a dominant technique, as in the Cambridge system or VADIS. IMS provides an enormous range of facilities; 50% of the code was said to be for recovery purposes [INFO75], although a more recent source stated that this figure was around 17% in 1978 (J.N. Gray of IBM Research Laboratory, San Jose, Calif., supplied this information in a private communication in February 1978). However, the application programmer, it seems, needs to build his own mechanisms and utilities, certainly if high integrity is required. The programmer also has to make explicit checkpoints if they are required.

The loss of any completed update can also be avoided by using careful replacement or multiple copies as in GEORGE 3, HIVE, CMIC and System R. It may also avoid the need for a salvation program (as in GEORGE 3). Also the differential files technique is very powerful and can be used to provide recovery facilities and crash recovery.

Audit trail with backup, or incremental dumping with backup, or audit trail with incremental dumping and backup, or multiple copies, are all techniques for recovery from more serious failures, which other recovery techniques cannot handle.

It is difficult to make a comparison be-

	System R [ASTR76] [LORI77]	*IMS* [IBM]	*GEORGE 3* [NEWE72]	*HIVE* [TAYI 76]	*MULTICS* [DALE65]	*Cambridge* [FRAS69]	*EMAS* [EMAS74]	*CMIC* [GIOR76]	*VADIS* [RAPP75]	*Newcastle* [VERH77a]
Salvation Program		•		•	•	•	•	•		
Incremental Dumping	•	•	•	•	•	•	•	•		
Audit Trail	•	•						•		
Differential Files								•		
Backup Current	•	•	•	•	•	•	•	•		•
Multiple Copies	•		•	•						
Careful Replacement			•					•		•

FIGURE 9. A cross-reference table of systems and recovery techniques.

tween costs and overheads for the various techniques. However, some general statements can be made:

- If failures do not occur often, the differential file and careful replacement techniques give extra overhead. The reason is that other recovery techniques, such as incremental dumping or backup/current version or a salvation program, are usually needed anyway for failures with which these two techniques cannot contend. However, the two techniques do cope with the particular failures in a much better way: the database is crash resistant, maintaining the correct state; it is more efficient because no separate tapes need to be mounted and processed.
- The multiple copies technique (e.g., in HIVE [TAYL76]) has very high overhead. Nonetheless, it meets HIVE's objective of very high integrity.
- The overhead with audit trails may be high, because every operation on the database may also necessitate an audit trail entry. This technique may be justified for recovery only if the audit trail is already required for certifying integrity or if it is absolutely required that almost all crashes are not catastrophic.
- The incremental dumping and backup/current version techniques are the best for recovery from highly damaging failures such as a head crash on disk. These techniques may not restore the correct state, but only a consistent state. The overhead of these techniques is tolerable, because their checkpoints are not too frequent.
- The cost of a salvation program completely depends on the number of crashes; overhead is accumulated only when the program is used.

The careful replacement technique is used increasingly in multiuser or multimachine environments [NEWE72, GAMB73, LAMP76, GIOR76, LORI77]. It is implied by the root-segment rule and leaf-first rule in systems using a hierarchy of devices [SCHW73]. The combination of careful replacement and multiple copies is also important [GIOR76, ASTR76, VERH77a]. The differential file technique [RAPP75, SEVE76] has many very nice features which have recently received much attention.

The attention that has been paid to these techniques during the last few years makes it reasonable to assume that they will be used more widely in the future. The techniques ensure that data is unlikely to be lost through failures. The cost of data integrity is lower, and its value higher, than a number of years ago. Data integrity is becoming a more important issue than efficiency.

ACKNOWLEDGMENTS

I would like to thank Brian Randell for insisting that I document the results of my search for information about current techniques for recovery in existing systems, and for his many very useful suggestions and comments.

I would also like to thank Stephen Todd, John Owlett (both of IBM, Peterlee, England), Ellis Cohen, Tom Anderson, Pete Lee and Phil Treleaven for reading earlier versions of this paper and giving me useful comments and suggestions. I appreciate Jim Gray's careful review of the manuscript and current information on System R.

My thanks are also due to the referees and to Peter J. Denning, the Editor-in-Chief of COMPUTING SURVEYS, for their useful comments and suggestions.

REFERENCES

[ANDE75] ANDERSON, T. *Provably safe programs*, Tech. Rep. 70, Computing Laboratory, Univ. Newcastle upon Tyne, UK, Feb. 1975.

[ASTR76] ASTRAHAN, M. M., et al. "System R: Relational approach to data base management," in *ACM Trans. Data Base Syst.* **1**, 2 (June 1976), 97–137.

[BANA77] BANATRE, J. P.; AND SHRIVASTAVA, S. K. *Reliable resource allocation between unreliable processes*, Tech. Rep. 99, Computing Laboratory, Univ. Newcastle upon Tyne, UK, April 1977.

[BJOR72] BJORK, L. A.; AND DAVIES, C. T. *The semantics of the presentation and recovery of integrity in a data base system*, IBM Tech. Rep. TR 02.540, Dec. 1972.

[BJOR75] BJORK, L. A. "Generalised audit trail requirements and concepts for data base applications," *IBM Syst J.* **14**, 3 (1975), 229–245.

[CURT77] CURTICE, R. M. "Integrity in data base systems," *Datamation* **23**, 5 (May 1977), 64–68.

[DALE65] DALEY, D. C.; AND NEUMANN, P. G. "A general purpose file system for secondary storage," in *1965 AFIPS Fall Jt. Computer Conf.*, Vol. 27, Part 1, Spartan Books, Washington, D.C., pp. 213–229.

[DAVI72] DAVIES, C. T. *A recovery/integrity ar-*

chitecture for a data system, IBM Tech. Rep. TR 02.528, May 1972.

[DENN76] DENNING, P. J. "Fault-tolerant operating systems," *Comput. Surv.* **8**, 4 (Dec. 1976), 359–389.

[EMAS74] REES, D. J. *The EMAS directory*, (EMAS report 2), MILLARD, G. E; REES, D. J.; AND WHITFIELD, H. *The standard EMAS subsystem*, (EMAS report 3); SHELNESS, N. A.; STEPHENS P. D.; AND WHITFIELD, H. *The Edinburgh multi-access system scheduling and allocation procedures in the resident supervisor*, (EMAS report 4); Dept. Computer Science, Univ. Edinburgh, Edinburgh, UK, April 1974.

[FRAS69] FRASER, A. G. "Integrity of a mass storage filing system," *Comput. J.* **12**, 1 (Feb. 1969), 1–5.

[GAMB73] GAMBLE, J N. "A file storage system for a multi-machine environment," PhD Thesis, Victoria Univ., Manchester, UK, Oct. 1973.

[GIOR76] GIORDANO, N. J.; AND SCHWARTZ, M. S. "Data base recovery at CMIC," in *Proc. 1976 SIGMOD Int. Conf. on Management of Data*, ACM, New York, pp. 33–42.

[GRAY70] GRAY, J. N. "Locking," in *Record of the Project MAC Conf. on Concurrent Systems and Parallel Computation*, 1970, Jack Dennis (Ed.), ACM, New York, pp. 97–112.

[GRAY76] GRAY, J. N.; LORIE, R. A.; PUTZOLU, G. R.; AND TRAIGER, J. L. "Granularity of locks and degrees of consistency in a shared data base," in *Modelling in data base management systems*, G. M. Nijssen (Ed.), Elsevier North-Holland, Inc., New York, 1976, pp. 365–394

[GRAY77] GRAY, J. N. "Notes on data base operating systems," in *Advanced course on operating systems*, Technical Univ. Munich, 1977, Elsevier North-Holland, Inc., New York.

[HOAR74] HOARE, C. A. R. "Monitors: an operating system structuring concept," *Commun. ACM* **17**, 10 (Oct 1974), 549–557.

[IBM] IBM Information Management System reference manuals. *IMS/VS, Utilities reference manual*, SH20-9029; *IMS/VS, Operators reference manual*, SH20-9028; *IMS/VS, System programmer reference manual*, SH20-9027, IBM, White Plains, N.Y.

[INFO75] *Infotech state of the art report: data base systems*, Infotech Information Ltd., Maidenhead, UK, 1975.

[KNUT73] KNUTH, D. E. *The art of computer programming, Vol. 3: sorting and searching*, Addison-Wesley Publ. Co., Reading, Mass., 1973.

[LAMP76] LAMPSON, B.; AND STURGIS, H. *Crash recovery in a distributed data storage system*, Computer Science Laboratory, Xerox Palo Alto Research Center, Palo Alto, Calif, 1976

[LIND76] LINDEN, T. A. "Operating system structures to support security and reliable software," *Comput. Surv* **8**, 4 (Dec. 1976), 409–445

[LOCK68] LOCKEMANN, P. C.; AND KNUTSEN, W D. "Recovery of disk contents after system failure," *Commun. ACM* **11**, 8 (Aug. 1968), 542.

[LORI77] LORIE, R. A. "Physical integrity in a large segmented database," *ACM Trans Database Syst.* **2**, 1 (March 1977), 91–104

[MART76] MARTIN, J *Principles of data-base management*, Prentice-Hall, Inc., Englewood Cliffs, N J., 1976, p. 4.

[MASC71] MASCALL, A. J *Studies of the reliability and performance of computing systems at Barclays Bank*, Internal memo SRM/10, Computing Laboratory, Univ. Newcastle upon Tyne, UK, 1971

[MASC73] MASCALL, A. J. *Checkpoint, backup and restart in a "reliable" system*, Internal memo SRM/37, Computing Laboratory, Univ. Newcastle upon Tyne, UK, April 1973.

[MELL77] MELLIAR-SMITH, P. M.; AND RANDELL, B. "Software reliability: the role of programmed exception handling," in *Proc. ACM Conf. on Language Design for Reliable Software*, 1977, ACM, New York, pp. 95–100.

[NEWE72] NEWELL, G. B. *Security and resilience in large scale operating systems*, 1900 Series Operating Systems Division, International Computers Ltd, London, 1972.

[RAND70] RANDELL, B. *Visit to BOAC*, Internal memo SRM/5, Computing Laboratory, Univ. Newcastle upon Tyne, UK, 1970

[RAND75] RANDELL, B. "System structure for software fault tolerance," *IEEE Trans. Softw. Eng.* **SE-1**, 2 (June 1975), 220–232.

[RAPP75] RAPPAPORT, R. L. "File structure design to facilitate on-line instantaneous updating," in *Proc. 1975 ACM SIGMOD Conf.*, ACM, New York, pp. 1–14.

[RAND78] RANDELL, B., LEE, P. A; AND TRELEAVEN, P. C. "Reliability issues in computing system design," see pp 123–165 this issue *Comput Surv*

[RUSS77] RUSSELL, D. L. "Process backup in producer-consumer systems," in *Proc. Symp. on Operating Systems Principles 1977; Operating Syst. Rev.* **11**, 5 (1977), 151–157.

[SCHW73] SCHWARTZ, M S. "A storage hierarchical addressing space for a computer file system," PhD Thesis, Case Western Univ., Cleveland, Ohio, 1973

[SEVE76] SEVERANCE, D. G.; AND LOHMAN, G. M. "Differential files: their application to the maintenance of large databases," *ACM Trans. Database Syst.* **1**, 3 (Sept. 1976), 256–267.

[SKLA76] SKLAROFF, J. R. "Redundancy management technique for space shuttle computers," *IBM J. Res. Dev* **20**, 1 (Jan 1976), 20–28.

[SMIT72] SMITH, J. L.; AND HOLDEN, T. S. "Restart of an operating system having a permanent file structure," *Comput. J* **15**, 1 (1972), 25–32.

[STER74] STERN, J. A. *Backup and recovery of on-line information in a computer utility*, Report MAC-TR-116, Project MAC, MIT, Cambridge, Mass., Jan. 1974

[TAYL76] TAYLOR, J. M. *Redundancy and recovery in the HIVE virtual machine*, Rep no. 76010, Royal Sıgnal and Radar Establishment, Christchurch, UK, May 1976.

[TITM74] TITMAN, P. J "An experimental data base system usıng binary relations," ın *Data base management*, J. W. Klimbıe, and K. L. Koffeman (Eds.), Elsevier North-Holland, Inc., New York, 1974, pp. 351-360.

[TONI75] TONIK, A. B. "Checkpoınt, restart and recovery: Selected annotated bibliography," *FDT, Bull. ACM SIGMOD* **7**, 3-4 (1975), 72-76.

[VERH77a] VERHOFSTAD, J. S. M. "Recovery and crash resıstance ın a filıng system," in *Proc 1977 ACM SIGMOD Int Conf on Management of Data*, ACM, New York, pp. 158-167.

[VERH77b] VERHOFSTAD, J. S. M "The constructıon of recoverable multi-level systems," PhD Thesis, Univ. Newcastle upon Tyne, UK, August 1977.

[WIMB71] WIMBROW, J. H "A large-scale interactıve admınistratıve system," *IBM Syst J.* **10**, 4 (1971), 260-282.

[WILK75] WILKES, M. V. *Tıme sharıng computer systems*, Elsevier North-Holland, Inc., New York, 1975

RECEIVED SEPTEMBER 8, 1977; FINAL REVISION ACCEPTED MARCH 7, 1978