Principles of Transaction-Oriented Database Recovery

THEO HAERDER

Fachbereich Informatik, University of Kaiserslautern, West Germany

ANDREAS REUTER[1]

IBM Research Laboratory, San Jose, California 95193

In this paper, a terminological framework is provided for describing different transaction-oriented recovery schemes for database systems in a conceptual rather than an implementation-dependent way. By introducing the terms materialized database, propagation strategy, and checkpoint, we obtain a means for classifying arbitrary implementations from a unified viewpoint. This is complemented by a classification scheme for logging techniques, which are precisely defined by using the other terms. It is shown that these criteria are related to all relevant questions such as speed and scope of recovery and amount of redundant information required. The primary purpose of this paper, however, is to establish an adequate and precise terminology for a topic in which the confusion of concepts and implementational aspects still imposes a lot of problems.

Categories and Subject Descriptors: D.4.5 [**Operating Systems**]: Reliability—*fault tolerance*; H.1.0 [**Models and Principles**]: General; H.2.2 [**Database Management**]: Physical Design—*recovery and restart*; H.2.4 [**Database Management**]: Systems—*transaction processing*; H.2.7 [**Database Management**]: Database Administration—*logging and recovery*

General Terms: Databases, Fault Tolerance, Transactions

INTRODUCTION

Database technology has seen tremendous progress during the past ten years. Concepts and facilities that evolved in the single-user batch environments of the early days have given rise to efficient multiuser database systems with user-friendly interfaces, distributed data management, etc. From a scientific viewpoint, database systems today are established as a mature discipline with well-approved methods and

technology. The methods and technology of such a discipline should be well represented in the literature by systematic surveys of the field. There are, in fact, a number of recent publications that attempt to summarize what is known about different aspects of database management [e.g., Astrahan et al. 1981; Stonebraker 1980; Gray et al. 1981; Kohler 1981; Bernstein and Goodman 1981; Codd 1982]. These papers fall into two categories: (1) descriptions of innovative prototype systems and (2) thorough analyses of special problems and their solutions, based on a clear methodological and terminological framework. We are con-

---
[1] Permanent address: Fachbereich Informatik, University of Kaiserslautern, West Germany.

CONTENTS

tributing to the second category in the field of database recovery. In particular, we are establishing a systematic framework for establishing and evaluating the basic concepts for fault-tolerant database operation.

The paper is organized as follows. Section 1 contains a short description of what recovery is expected to accomplish and which notion of consistency we assume. This involves introducing the transaction, which has proved to be the major paradigm for synchronization and recovery in advanced database systems. This is also the most important difference between this paper and Verhofstadt's survey, in which techniques for file recovery are described without using a particular notion of consistency [Verhofstadt 1978]. Section 2 provides an implementational model for database systems, that is, a mapping hierarchy of data types. Section 3 introduces the key concepts of our framework, describing the database states after a crash, the type of log information required, and additional measures for facilitating recovery. Crash

recovery is demonstrated with three sample implementation techniques. Section 4 applies concepts addressed in previous sections on media recovery, and Section 5 summarizes the scope of our taxonomy.

1. DATABASE RECOVERY: WHAT IT IS EXPECTED TO DO

Understanding the concepts of database recovery requires a clear comprehension of two factors:

- the type of failure the database has to cope with, and
- the notion of consistency that is assumed as a criterion for describing the state to be reestablished.

Before beginning a discussion of these factors, we would like to point out that the contents of this section rely on the description of failure types and the concept of a transaction given by Gray et al. [1981].

1.1 What Is a Transaction?

It was observed quite early that manipulating data in a multiuser environment requires some kind of isolation to prevent uncontrolled and undesired interactions. A user (or process) often does things when working with a database that are, up to a certain point in time, of tentative or preliminary value. The user may read some data and modify others before finding out that some of the initial input was wrong, invalidating everything that was done up to that point. Consequently, the user wants to remove what he or she has done from the system. If other users (or processes) have already seen the "dirty data" [Gray et al. 1981] and made decisions based upon it, they obviously will encounter difficulties. The following questions must be considered:

- How do they get the message that some of their input data has disappeared, when it is possible that they have already finished their job and left the terminal?
- How do they cope with such a situation? Do they also throw away what they have done, possibly affecting others in turn? Do they reprocess the affected parts of their program?

```
FUNDS_TRANSFER: PROCEDURE;
  $BEGIN_TRANSACTION;
  ON ERROR DO;                                        /*in case of error*/
    $RESTORE_TRANSACTION;                             /*undo all work*/
    GET INPUT MESSAGE;                                /*reacquire input*/
    PUT MESSAGE ('TRANSFER FAILED');                  /*report failure*/
    GO TO COMMIT;
    END;
  GET INPUT MESSAGE;                                  /*get and parse input*/
  EXTRACT ACCOUNT_DEBIT, ACCOUNT_CREDIT,
    AMOUNT FROM MESSAGE;
  $UPDATE ACCOUNTS                                    /*do debit*/
    SET BALANCE = BALANCE - AMOUNT
      WHERE ACCOUNTS NUMBER = ACCOUNTS_DEBIT;
  $UPDATE ACCOUNTS                                    /*do credit*/
    SET BALANCE = BALANCE + AMOUNT
      WHERE ACCOUNTS NUMBER = ACCOUNTS_CREDIT;
  $INSERT INTO HISTORY                                /*keep audit trail*/
    ⟨DATE, MESSAGE⟩;
  PUT MESSAGE ('TRANSFER DONE');                      /*report success*/
 COMMIT:                                              /*commit updates*/
$COMMIT_TRANSACTION;
  END;                                                /*end of program*/
```

Figure 1. Example of a transaction program. (From Gray et al. [1981].)

These situations and dependencies have been investigated thoroughly by Bjork and Davies in their studies of the so-called "spheres of control" [Bjork 1973; Davies 1973, 1978]. They indicate that data being operated by a process must be isolated in some way that lets others know the degree of reliability provided for these data, that is,

- Will the data be changed without notification to others?
- Will others be informed about changes?
- Will the value definitely not change any more?

This ambitious concept was restricted to use in database systems by Eswaran et al. [1976] and given its current name, the "transaction." The transaction basically reflects the idea that the activities of a particular user are isolated from all concurrent activities, but restricts the degree of isolation and the length of a transaction. Typically, a transaction is a short sequence of interactions with the database, using operators such as FIND a record or MODIFY an item, which represents one meaningful activity in the user's environment. The standard example that is generally used to explain the idea is the transfer of money from one account to another. The corresponding transaction program is given in Figure 1.

The concept of a transaction, which includes all database interactions between $BEGIN_TRANSACTION and $COMMIT_TRANSACTION in the above example, requires that all of its actions be executed *indivisibly*: Either all actions are properly reflected in the database or nothing has happened. No changes are reflected in the database if at any point in time before reaching the $COMMIT_TRANSACTION the user enters the ERROR clause containing the $RESTORE_TRANSACTION. To achieve this kind of indivisibility, a transaction must have four properties:

Atomicity. It must be of the all-or-nothing type described above, and the user must, whatever happens, know which state he or she is in.

Consistency. A transaction reaching its normal end (EOT, end of transaction), thereby committing its results, preserves the consistency of the database. In other words, each successful transaction by definition commits only legal results. This con-

BEGIN	BEGIN	BEGIN
READ	READ	READ
WRITE	WRITE	WRITE
READ	READ	READ
⋮	⋮	⋮
WRITE	ABORT	⇐SYSTEM ABORTS
COMMIT		TRANSACTION

Figure 2. Three possible outcomes of a transaction. (From Gray et al. [1981].)

dition is necessary for the fourth property, durability.

Isolation. Events within a transaction must be hidden from other transactions running concurrently. If this were not the case, a transaction could not be reset to its beginning for the reasons sketched above. The techniques that achieve isolation are known as *synchronization*, and since Gray et al. [1976] there have been numerous contributions to this topic of database research [Kohler 1981].

Durability. Once a transaction has been completed and has committed its results to the database, the system must guarantee that these results survive any subsequent malfunctions. Since there is no sphere of control constituting a set of transactions, the database management system (DBMS) has no control beyond transaction boundaries. Therefore the user must have a guarantee that the things the system says have happened have actually happened. Since, by definition, each transaction is correct, the effects of an inevitable incorrect transaction (i.e., the transaction containing faulty data) can only be removed by countertransactions.

These four properties, atomicity, consistency, isolation, and durability (ACID), describe the major highlights of the transaction paradigm, which has influenced many aspects of development in database systems. We therefore consider the question of whether the transaction is supported by a particular system to be the ACID test of the system's quality.

In summary, a transaction can terminate in the three ways illustrated in Figure 2. It is hoped that the transaction will reach its commit point, yielding the all case (as in the all-or-nothing dichotomy). Sometimes the transaction detects bad input or other violations of consistency, preventing a normal termination, in which case it will reset all that it has done (abort). Finally, a transaction may run into a problem that can only be detected by the system, such as time-out or deadlock, in which case its effects are aborted by the DBMS.

In addition to the above events occurring during normal execution, a transaction can also be affected by a system crash. This is discussed in the next section.

1.2 Which Failures Have to Be Anticipated

In order to design and implement a recovery component, one must know precisely which types of failures are to be considered, how often they will occur, how much time is expected for recovery, etc. One must also make assumptions about the reliability of the underlying hardware and storage media, and about dependencies between different failure modes. However, the list of anticipated failures will never be complete for these reasons:

• For each set of failures that one can think of, there is at least one that was forgotten.
• Some failures are extremely rare. The cost of redundancy needed to cope with them may be so high that it may be a sensible design decision to exclude these failures from consideration. If one of them does occur, however, the system will not be able to recover from the situation automatically, and the database will be corrupted. The techniques for handling this catastrophe are beyond the scope of this paper.

We shall consider the following types of failure:

Transaction Failure. The transaction of failure has already been mentioned in the previous section. For various reasons, the transaction program does not reach its normal commit and has to be reset back to its beginning, either at its own request or on behalf of the DBMS. Gray indicates that 3 percent of all transactions terminate abnormally, but this rate is not likely to be a constant [Gray et al. 1981]. From our own experiences with different application da-

tabases, and from Gray's result [Effelsberg et al. 1981; Gray 1981], we can conclude that

- Within one application, the ratio of transactions that abort themselves is rather constant, depending only on the amount of incorrect input data, the quality of consistency checking performed by the transaction program, etc.
- The ratio of transactions being aborted by the DBMS, especially those caused by deadlocks, depends to a great extent on the degree of parallelism, the granularity of locking used by the DBMS, the logical schema (there may be hot spot data, or data that are very frequently referenced by many concurrent transactions), and the degree of interference between concurrent activities (which is, in turn, very application dependent).

For our classification, it is sufficient to say that transaction failures occur *10–100 times per minute*, and that recovery from these failures must take place within the time required by the transaction for its regular execution.

System Failure. The system failures that we are considering can be caused by a bug in the DBMS code, an operating system fault, or a hardware failure. In each of these cases processing is terminated in an uncontrolled manner, and we assume that the contents of main memory are lost. Since database-related secondary (nonvolatile) storage remains unaffected, we require that a recovery take place in the same amount of time that would have been required for the execution of all interrupted transactions. If one transaction is executed within the order of 10 milliseconds to 1 second, the recovery should take no more than a few minutes. A system failure is assumed to occur *several times a week*, depending on the stability of both the DBMS and its operational environment.

Media Failure. Besides these more or less normal failures, we have to anticipate the loss of some or all of the secondary storage holding the database. There are several causes for such a problem, the most common of which are

- bugs in the operating system routines for writing the disk,
- hardware errors in the channel or disk controller,
- head crash,
- loss of information due to magnetic decay.

Such a situation can only be overcome by full redundancy, that is, by a copy of the database and an audit trail covering what has happened since then.

Magnetic storage devices are usually very reliable, and recovery from a media failure is not likely to happen more often than *once or twice a year.* Depending on the size of a database, the media used for storing the copy, and the age of the copy, recovery of this type will take on the order of 1 hour.

1.3 Summary of Recovery Actions

As we mentioned in Section 1.1, the notion of consistency that we use for defining the targets of recovery is tied to the transaction paradigm, which we have encapsulated in the "ACID principle." According to this definition, a database is consistent *if and only if* it contains the results of successful transactions. Such a state will hereafter be called *transaction consistent* or *logically consistent.* A transaction, in turn, must not see anything but effects of complete transactions (i.e., a consistent database in those parts that it uses), and will then, by definition, create a consistent update of the database. What does that mean for the recovery component?

Let us for the moment ignore transactions being aborted during normal execution and consider only a system failure (a crash). We might then encounter the situation depicted in Figure 3. Transactions T1, T2, and T3 have committed before the crash, and therefore will survive. Recovery after a system failure must ensure that the effects of all successful transactions are actually reflected in the database. But what is to be done with T4 and T5? Transactions have been defined to be atomic; they either succeed or disappear as though they had never been entered. There is therefore no choice about what to do after a system
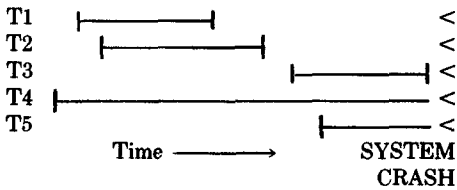
Figure 3. Scenario for discussing transaction-oriented recovery. (From Gray et al. [1981].)

failure; the effects of all incomplete transactions must be removed from the database. Clearly, a recovery component adhering to these principles will produce a transaction-consistent database. Since *all* successful transactions have contributed to the database state, it will be the *most recent* transaction-consistent state. We now can distinguish four recovery actions coping with different situations [Gray 1978]:

Transaction UNDO. If a transaction aborts itself or must be aborted by the system during normal execution, this will be called "transaction UNDO." By definition, UNDO removes all effects of this transaction from the database and does not influence any other transaction.

Global UNDO. When recovering from a system failure, the effects of all incomplete transactions have to be rolled back.

Partial REDO. When recovering from a system failure, since execution has been terminated in an uncontrolled manner, results of complete transactions may not yet be reflected in the database. Hence they must be repeated, if necessary, by the recovery component.

Global REDO. Gray terms this recovery action "archive recovery" [Gray et al. 1981]. The database is assumed to be physically destroyed; we therefore must start from a copy that reflects the state of the database some days, weeks, or months ago. Since transactions are typically short, we need not consider incomplete transactions over such a long time. Rather we have to supplement the copy with the effects of all transactions that have committed since the copy was created.

With these definitions we have introduced the transaction as the *only unit of recovery* in a database system. This is an ideal condition that does not exactly match

reality. For example, transactions might be nested, that is, composed of smaller subtransactions. These subtransactions also are atomic, consistent, and isolated—but they are not durable. Since the results of subtransactions are removed whenever the enclosing transaction is undone, durability can only be guaranteed for the highest transaction in the composition hierarchy. A two-level nesting of transactions can be found in System R, in which an arbitrary number of save points can be generated inside a transaction [Gray et al. 1981]. The database and the processing state can be reset to any of these save points by the application program.

Another extension of the transaction concept is necessary in fields like CAD. Here the units of consistent state transitions, that is, the design steps, are so long (days or weeks) that it is not feasible to treat them as indivisible actions. Hence these *long* transactions are consistent, isolated, and durable, but they are not atomic [Gray 1981]. It is sufficient for the purpose of our taxonomy to consider "ideal" transactions only.

2. THE MAPPING HIERARCHY OF A DBMS

There are numerous techniques and algorithms for implementing database recovery, many of which have been described in detail by Verhofstadt [1978]. We want to reduce these various methods to a small set of basic concepts, allowing a simple, yet precise classification of all reasonable implementation techniques; for the purposes of illustration, we need a basic model of the DBMS architecture and its hardware environment. This model, although it contains many familiar terms from systems like INGRES, System R, or those of the CODASYL [1973, 1978] type, is in fact a rudimentary database architecture that can also be applied to unconventional approaches like CASSM or DIRECT [Smith and Smith 1979], although this is not our purpose here.

2.1 The Mapping Process: Objects and Operations

The model shown in Table 1 describes the major steps of dynamic abstraction from the level of physical storage up to the user

Table 1. Description of the DB-Mapping Hierarchy

Level of abstraction	Objects	Auxiliary mapping data
Nonprocedural or algebraic access	Relations, views tuples	Logical schema description
Record-oriented, navigational access	Records, sets, hierarchies, networks	Logical and physical schema description
Record and access path management	Physical records, access paths	Free space tables, DB-key translation tables
Propagation control	Segments, pages	Page tables, Bloom filters
File management	Files, blocks	Directories, VTOCs, etc.

interface. At the bottom, the database consists of some billions of bits stored on disk, which are interpreted by the DBMS into meaningful information on which the user can operate. With each level of abstraction (proceeding from the bottom up), the objects become more complex, allowing more powerful operations and being constrained by a larger number of integrity rules. The uppermost interface supports one of the well-known data models, whether relational, networklike, or hierarchical.

Note that this mapping hierarchy is virtually contained in each DBMS, although for performance reasons it will hardly be reflected in the module structure. We shall briefly sketch the characteristics of each layer, with enough detail to establish our taxonomy. For a more complete description see Haerder and Reuter [1983].

File Management. The lowest layer operates directly on the bit patterns stored on some nonvolatile, direct access device like a disk, drum, or even magnetic bubble memory. This layer copes with the physical characteristics of each storage type and abstracts these characteristics into fixed-length blocks. These blocks can be read, written, and identified by a (relative) block number. This kind of abstraction is usually done by the data management system (DMS) of a normal general-purpose operating system.

Propagation[2] Control. This level is not usually considered separately in the current

database literature, but for reasons that will become clear in the following sections we strictly distinguish between *pages* and *blocks.* A page is a fixed-length partition of a linear address space and is mapped into a physical block by the propagation control layer. Therefore a page can be stored in different blocks during its lifetime in the database, depending on the strategy implemented for propagation control.

Access Path Management. This layer implements mapping functions much more complicated than those performed by subordinate layers. It has to maintain all physical object representations in the database (records, fields, etc.), and their related access paths (pointers, hash tables, search trees, etc.) in a *potentially unlimited* linear virtual address space. This address space, which is divided into fixed-length pages, is provided by the upper interface of the supporting layer. For performance reasons, the partitioning of data into pages is still visible on this level.

Navigational Access Layer. At the top of this layer we find the operations and objects that are typical for a procedural data manipulation language (DML). Occurrences of record types and members of sets are handled by statements like STORE, MODIFY, FIND NEXT, and CONNECT [CODASYL 1978]. At this interface, the user navigates one record at a time through a hierarchy, through a network, or along logical access paths.

Nonprocedural Access Layer. This level provides a nonprocedural interface to the

[2] This term is introduced in Section 2.4; its meaning is not essential to the understanding of this paragraph.
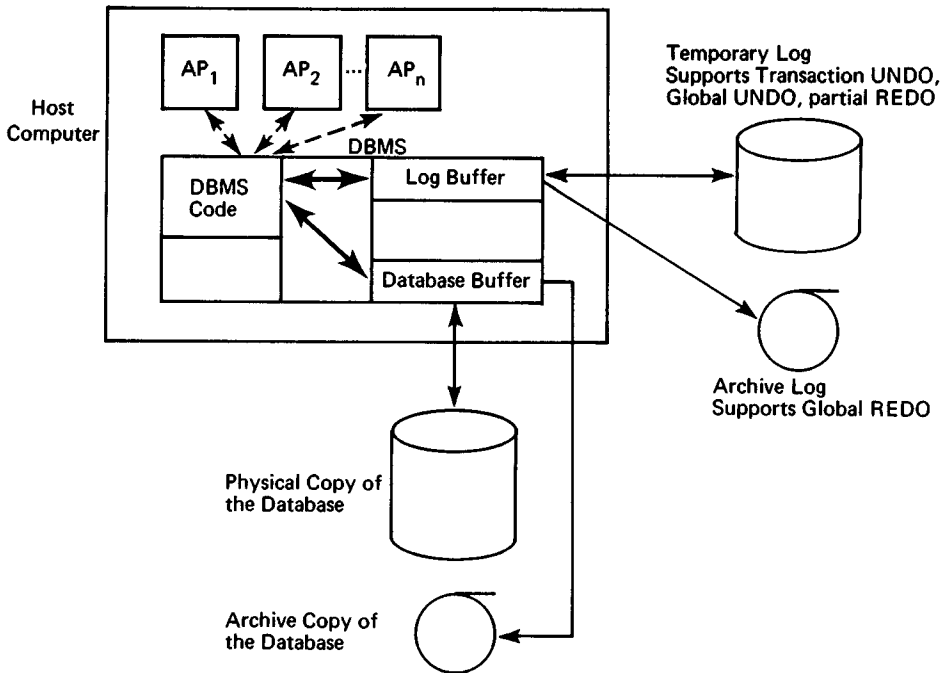
Figure 4. Storage hierarchy of a DBMS during normal mode of operation.

database. With each operation the user can handle sets of results rather than single records. A relational model with high-level query languages like SQL or QUEL is a convenient example of the abstraction achieved by the top layer [Chamberlin 1980; Stonebraker et al. 1976].

On each level, the mapping of higher objects to more elementary ones requires additional data structures, some of which are shown in Table 1.

2.2 The Storage Hierarchy: Implementational Environment

Both the number of redundant data required to support the recovery actions described in Section 1 and the methods of collecting such data are strongly influenced by various properties of the different storage media used by the DBMS. In particular, the dependencies between volatile and permanent storage have a strong impact on algorithms for gathering redundant information and implementing recovery measures [Chen 1978]. As a descriptive framework we shall use a storage hierarchy, as

shown in Figure 4. It closely resembles the situation that must be dealt with by most of today's commercial database systems.

The host computer, where the application programs and DBMS are located, has a main memory, which is usually volatile.[3] Hence we assume that the contents of the database buffer, as well as the contents of the output buffers to the log files, are lost whenever the DBMS terminates abnormally. Below the volatile main memory there is a two-level hierarchy of permanent copies of the database. One level contains an on-line version of the database in direct access memory; the other contains an archive copy as a provision against loss of the on-line copy. While both are functionally situated on the same level, the on-line copy is almost always up-to-date, whereas the archive copy can contain an old state of the database. Our main concern here is database recovery, which, like all provisions for

---

[3] In some real-time applications main memory is supported by a battery backup. It is possible that in the future mainframes will have some stable buffer storage. However, we are not considering these conditions here.

fault tolerance, is based upon redundancy. We have mentioned one type of redundancy: the archive copy, kept as a starting point for reconstruction of an up-to-date on-line version of the database (global REDO). This is discussed in more detail in Section 4. To support this, and other recovery actions introduced in Section 1, two types of log files are required:

Temporary Log. The information collected in this file supports crash recovery; that is, it contains information needed to reconstruct the most recent database (DB) buffer. Selective transaction UNDO requires random access to the log records. Therefore we assume that the temporary log is located on disk.

Archive Log. This file supports global REDO after a media failure. It depends on the availability of the archive copy and must contain all changes committed to the database after the state reflected in the archive copy. Since the archive log is always processed in sequential order, we assume that the archive log is written on magnetic tape.

2.3 Different Views of a Database

In Section 2.1, we indicated that the database looks different at each level of abstraction, with each level using different objects and interfaces. But this is not what we mean by "different views of a database" in this section. We have observed that the process of abstraction really begins at Level 3, up to which there is only a more convenient representation of data in external storage. At this level, abstraction is dependent on which pages actually establish the linear address space, that is, which block is read when a certain page is referenced. In the event of a failure, there are different possibilities for retrieving the contents of a page. These possibilities are denoted by different views of the database:

The *current database* comprises all objects accessible to the DBMS during normal processing. The current contents of all pages can be found on disk, except for those pages that have been recently modified. Their new contents are found in the DB buffer. The mapping hierarchy is completely correct.

The *materialized database* is the state that the DBMS finds at restart after a crash *without* having applied any log information. There is no buffer. Hence some page modifications (even of successful transactions) may not be reflected in the on-line copy. It is also possible that a new state of a page has been written to disk, but the control structure that maps pages to blocks has not yet been updated. In this case, a reference to such a page will yield the old value. This view of the database is what the recovery system has to transform into the most recent logically consistent current database.

The *physical database* is composed of all blocks of the on-line copy containing page images—current or obsolete. Depending on the strategy used on Level 2, there may be different values for one page in the physical database, none of which are necessarily the current contents. This view is not normally used by recovery procedures, but a salvation program would try to exploit all information contained therein.

With these views of a database, we can distinguish three types of update operations—all of which explain the mapping function provided by the propagation control level. First, we have the *modification of page contents* caused by some higher level module. This operation takes place in the DB buffer and therefore affects only the current database. Second, there is the *write* operation, transferring a modified page to a block on disk. In general, this affects only the physical database. If the information about the block containing the new page value is stored in volatile memory, the new contents will not be accessible after a crash; that is, it is not yet part of the materialized database. The operation that makes a previously written page image part of the materialized database is called *propagation*. This operation writes the updated control structures for mapping pages to blocks in a safe, nonvolatile place, so that they are available after a crash.

If pages are always written to the same block (the so-called "update-in-place" operation, which is done in most commercial DBMS), writing implicitly is the equivalent

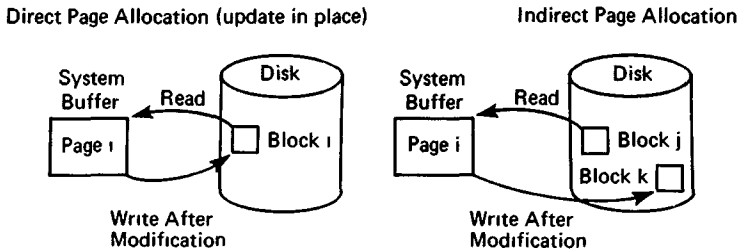Direct Page Allocation (update in place)   Indirect Page Allocation

Figure 5. Page allocation principles.

of propagation. However, there is an important difference between these operations if a page can be stored in different blocks. This is explained in the next section.

2.4 Mapping Concepts for Updates

In this section, we define a number of concepts related to the operation of mapping changes in a database from volatile to nonvolatile storage. They are directly related to the views of a database introduced previously. The key issue is that each modification of a page (which changes the current database) takes place in the database buffer and is allocated to *volatile* storage. In order to save this state, the corresponding page must be brought to nonvolatile storage, that is, to the physical database. Two different schemes for accomplishing this can be applied, as sketched in Figure 5.

With *direct page allocation*, each page of a segment is related to exactly one block of the corresponding file. Each output of a modified page causes an update in place. By using an *indirect page allocation* scheme, each output is directed to a new block, leaving the old contents of the page unchanged. It provides the option of holding n successive versions of a page. The moment when a younger version definitively replaces an older one can be determined by appropriate (consistency-related) criteria; it is no longer bound to the moment of writing. This update scheme has some very attractive properties in case of recovery, as is shown later on. Direct page allocation leaves no choice as to when to make a new version part of the materialized database; the output operation destroys the previous image. Hence in this case writing and propagating coincide.

There is still another important difference between direct and indirect page allocation schemes, which can be characterized as follows:

• In *direct* page allocation, each single propagation (physical write) is interruptable by a system crash, thus leaving the materialized, and possibly the physical, database in an inconsistent state.
• In *indirect* page allocation, there is always a way back to the old state. Hence propagation of an arbitrary set of pages can be made uninterruptable by system crashes. References to such algorithms will be given.

On the basis of this observation, we can distinguish two types of propagation strategies:

ATOMIC. Any set of modified pages can be propagated as a unit, such that either all or none of the updates become part of the materialized database.

¬ATOMIC. Pages are written to blocks according to an update-in-place policy. Since no set of pages can be written indivisibly (even a single write may be interrupted somewhere in between), propagation is vulnerable to system crashes.

Of course, many details have been omitted from Figure 5. In particular, there is no hint of the techniques used to make propagation take place atomically in case of indirect page mapping. We have tried to illustrate aspects of this issue in Figure 6. Figure 6 contains a comparison of the current and the materialized database for the update-in-place scheme and three different implementations of indirect page mapping allowing for ATOMIC propagation. Figure 6b refers to the well-known shadow page
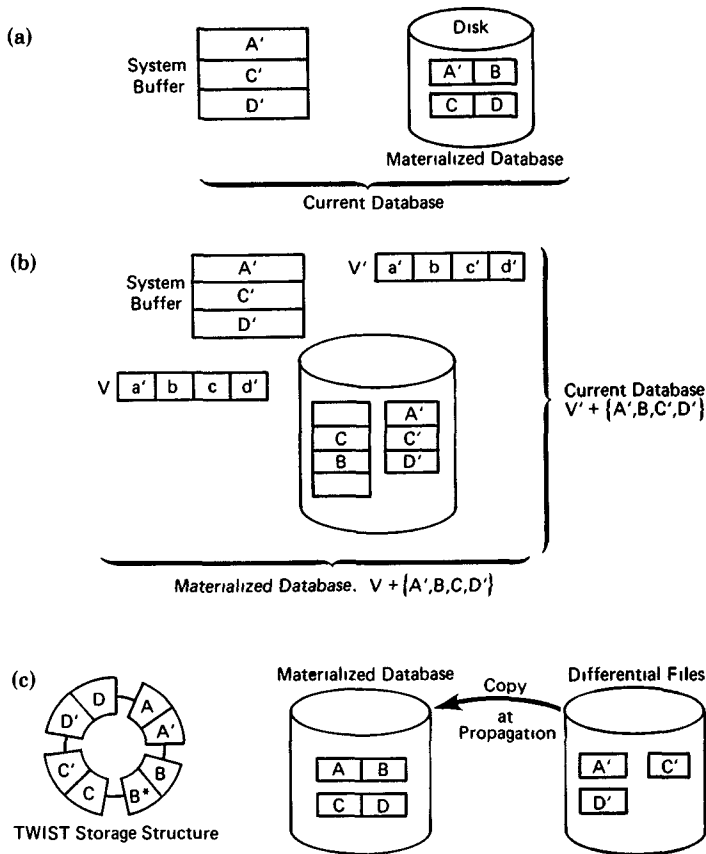
Figure 6. Current versus materialized database in ¬ATOMIC (a) and ATOMIC (b and c) propagation.

mechanism [Lorie 1977]. The mapping of page numbers to block numbers is done by using page tables. These tables have one entry per page containing the block number where the page contents are stored. The *shadow pages*, accessed via the shadow page Table V, preserve the old state of the materialized database. The current version is defined by the current page Table V'. Before this state is made stable (propagated), all changed pages are written to their new blocks, and so is the current page table. If this fails, the database will come up in its old state. When all pages have been written related to the new state, ATOMIC propagation takes place by changing one record on disk (which now points to V' rather than V) in a way that cannot be confused by a system crash. Thus the problem of indivisibly propagating a set of pages has

been reduced to safely updating one record, which can be done in a simple way. For details, see Lorie [1977].

There are other implementations for ATOMIC propagation. One is based on maintaining two recent versions of a page. For each page access, both versions have to be read into the buffer. This can be done with minimal overhead by storing them in adjacent disk blocks and reading them with chained I/O. The latest version, recognized by a time stamp, is kept in the buffer; the other one is immediately discarded. A modified page replaces the older version on disk. ATOMIC propagation is accomplished by incrementing a special counter that is related to the time stamps in the pages. Details can be found in Reuter [1980]. Another approach to ATOMIC propagation has been introduced under the name "dif-

ferential files" by Severance and Lohman [1976]. Modified pages are written to a separate (differential) file. Propagating these updates to the main database is not ATOMIC in itself, but once all modifications are written to the differential file, propagation can be repeated as often as wished. In other words, the process of copying modified pages into the materialized database can be made to appear ATOMIC. A variant of this technique, the "intention list," is described by Lampson and Sturgis [1979] and Sturgis et al. [1980].

Thus far we have shown that arbitrary sets of pages can be propagated in an ATOMIC manner using indirect page allocation. In the next section we discuss how these sets of pages for propagation should be defined.

3. CRASH RECOVERY

In order to illustrate the consequences of the concepts introduced thus far, we shall present a detailed discussion of crash recovery. First, we consider the state in which a database is left when the system terminates abnormally. From this we derive the type of redundant (log) information required to reestablish a transaction-consistent state, which is the overall purpose of DB recovery. After completing our classification scheme, we give examples of recovery techniques in currently available database systems. Finally, we present a table containing a qualitative evaluation of all instances encompassed by our taxonomy (Table 4).

Note that the results in this section also apply to transaction UNDO—a much simpler case of global UNDO, which applies when the DBMS is processing normally and no information is lost.

3.1 State of the Database after a Crash

After a crash, the DBMS has to restart by applying all the necessary recovery actions described in Section 1. The DB buffer is lost, as is the *current database*, the only view of the database to contain the most recent state of processing. Assuming that the on-line copy of the database is intact, there are the *materialized database* and the

temporary log file from which to start recovery. We have not discussed the contents of the log files for the reason that the type and number of log data to be written during normal processing are dependent upon the state of the materialized database after a crash. This state, in turn, depends upon which method of page allocation and propagation is used.

In the case of direct page allocation and ¬ATOMIC propagation, each write operation affects the materialized database. The decision to write pages is made by the *buffer manager* according to buffer capacity at points in time that appear arbitrary. Hence the state of the materialized database after a crash is unpredictable: When recent modifications are reflected in the materialized database, it is not possible (without further provisions) to know which pages were modified by complete transactions (whose contents must be reconstructed by partial REDO) and which pages were modified by incomplete transactions (whose contents must be returned to their previous state by global UNDO). Further possibilities for providing against this situation are briefly discussed in Section 3.2.1.

In the case of indirect page allocation and ATOMIC propagation, we know much more about the state of the materialized database after crash. ATOMIC propagation is indivisible by any type of failure, and therefore we find the materialized database to be exactly in the state produced by the most recent successful propagation. This state may still be inconsistent in that not all updates of complete transactions are visible, and some effects of incomplete transactions are. However, ATOMIC propagation ensures that a set of related pages is propagated in a safe manner by restricting propagation to points in time when the current database fulfills certain consistency constraints. When these constraints are satisfied, the updates can be mapped to the materialized database all at once. Since the current database is consistent in terms of the access path management level—where propagation occurs—this also ensures that all internal pointers, tree structures, tables, etc. are correct. Later on, we also discuss schemes that allow for transaction-consistent propagation.

The state of the materialized database after a crash can be summarized as follows:

¬*ATOMIC Propagation.* Nothing is known about the state of the materialized database; it must be characterized as "chaotic."

ATOMIC Propagation. The materialized database is in the state produced by the most recent propagation. Since this is bound by certain consistency constraints, the materialized database will be consistent (but not necessarily up-to-date) at least up to the third level of the mapping hierarchy.

In the case of ¬ATOMIC propagation, one cannot expect to read valid images for all pages from the materialized database after a crash; it is inconsistent on the propagation level, and all abstractions on higher levels will fail. In the case of ATOMIC propagation, the materialized database is consistent at least on Level 3, thus allowing for the execution of operations on Level 4 (DML statements).

3.2 Types of Log Information to Support Recovery Actions

The temporary log file must contain all the information required to transform the materialized database "as found" into the most recent transaction-consistent state (see Section 1). As we have shown, the materialized database can be in more or less defined states, may or may not fulfill consistency constraints, etc. Hence the number of log data will be determined by what is contained in the materialized database at the beginning of restart. We can be fairly certain of the contents of the materialized database in the case of ATOMIC propagation, but the result of ¬ATOMIC schemes have been shown to be unpredictable. There are, however, additional measures to somewhat reduce the degree of uncertainty resulting from ¬ATOMIC propagation, as discussed in the following section.

3.2.1 Dependencies between Buffer Manager and Recovery Component

3.2.1.1 Buffer Management and UNDO Recovery Actions. During the normal mode of operation, modified pages are written to disk by some replacement algorithm managing the database buffer. Ideally, this happens at points in time determined solely by buffer occupation and, from a consistency perspective, seem to be arbitrary. In general, even dirty data, that is, pages modified by incomplete transactions, may be written to the physical database. Hence the UNDO operations described earlier will have to recover the contents of both the materialized database and the external storage media. The only way to avoid this requires that the buffer manager be modified to prevent it from writing or propagating dirty pages under all circumstances. In this case, UNDO could be considerably simplified:

- If no dirty pages are propagated, global UNDO becomes virtually unnecessary that is, if there are no dirty data in the materialized database.
- If no dirty pages are written, transaction UNDO can be limited to main storage (buffer) operations.

The major disadvantage of this idea is that very large database buffers would be required (e.g., for long batch update transactions), making it generally incompatible with existing systems. However, the two different methods of handling modified pages introduced with this idea have important implications with UNDO recovery. We shall refer to these methods as:

STEAL. Modified pages may be written and/or propagated at any time.

¬*STEAL.* Modified pages are kept in buffer at least until the end of the transaction (EOT).

The definition of STEAL can be based on either writing or propagating, which are not discriminated in ¬ATOMIC schemes. In the case of ATOMIC propagation both variants of STEAL are conceivable, and each would have a different impact on UNDO recovery actions; in the case of ¬STEAL, no logging is required for UNDO purposes.

3.2.1.2 Buffer Management and REDO Recovery Actions. As soon as a transaction commits, all of its results must survive any subsequent failure (*durability*). Committed updates that have not been propagated to

the materialized database would definitely be lost in case of a system crash, and so there must be enough redundant information in the log file to reconstruct these results during restart (partial REDO). It is conceivable, however to avoid this kind of recovery by the following technique.

During Phase 1 of EOT processing all pages modified by this transaction are propagated to the materialized database; that is, their writing *and* propagation are enforced. Then we can be sure that either the transaction is complete, which means that all of its results are safely recorded (no partial REDO), or in case of a crash, some updates are not yet written, which means that the transaction is not successful and must be rolled back (UNDO recovery actions).

Thus we have another criterion concerning buffer handling, which is related to the necessity of REDO recovery during restart:

FORCE. All modified pages are written and propagated during EOT processing.

¬FORCE. No propagation is triggered during EOT processing.

The implications with regard to the gathering of log data are quite straightforward in the case of FORCE. No logging is required for *partial REDO*; in the case of ¬FORCE such information is required. While FORCE avoids partial REDO, there must still be some REDO-log information for *global REDO* to provide against loss of the on-line copy of the database.

3.2.2 Classification of Log Data

Depending on which of the write and propagation schemes introduced above are being implemented, we will have to collect log information for the purpose of

- removing invalid data (modifications effected by incomplete transactions) from the materialized database and
- supplementing the materialized database with updates of complete transactions that were not contained in it at the time of crash.

In this section, we briefly describe what such log data can look like and when such

Table 2. Classification Scheme for Log Data

	State	Transition
Logical	—	Actions (DML statements)
Physical	Before images After images	EXOR differences

data are applicable to the crash state of the materialized database.

Log data are redundant information, collected for the sole purpose of recovery from a crash or a media failure. They do not undergo the mapping process of the database objects, but are obtained on a certain level of the mapping hierarchy and written directly to nonvolatile storage, that is, the log files. There are two different, albeit not fully orthogonal, criteria for classifying log data. The first is concerned with the *type* of objects to be logged. If some part of the physical representation, that is, the bit pattern, is written to the log, we refer to it as *physical logging*; if the operators and their arguments are recorded on a higher level, this is called *logical logging*. The second criterion concerns whether the *state* of the database—before or after a change—or the *transition* causing the change is to be logged. Table 2 contains some examples for these different types of logging, which are explained below.

Physical State Logging on Page Level. The most basic method, which is still applied in many commercial DBMSs, uses the page as the unit of log information. Each time a part of the linear address space is changed by some modification, insertion, etc., the whole page containing this part of the linear address space is written to the log. If UNDO logging is required, this will be done before the change takes place, yielding the so-called *before image*. For REDO purposes, the resulting page state is recorded as an *after image*.

Physical Transition Logging on Page Level. This logging technique is based also on pages. However, it does not explicitly record the old and new *states* of a page; rather it writes the *difference between them* to the log. The function used for computing the "difference" between two bit strings is

the exclusive-or, which is both commutative and associative as required by the recovery algorithm. If this difference is applied to the old state of a page, again using the exclusive-or, the new state will result. On the other hand, applying it to the new state will yield the old state. There are some problems in the details of this approach, but these are beyond the scope of the paper.

The two methods of page logging that we have discussed can be compared as follows:

• Transition logging requires only one log entry (the difference), whereas state logging uses both a before image and an after image. If there are multiple changes applied to the same page during one transaction, transition logging can express these either by successive differences or by one accumulated difference. With state logging, the first before image and the last after image are required.

• Since there are usually only a small number of data inside a page affected by a change, the exclusive-or difference will contain long strings of 0's, which can be removed by well-known compression techniques. Hence transition logging can potentially require much less space than does state logging.

Physical State Logging on Access Path Level. Physical logging can also be applied to the objects of the access path level, namely, physical records, access path structures, tables, etc. The log component has to be aware of these storage structures and record only the changed entry, rather than blindly logging the whole page around it. The advantage of this requirement is obvious: By logging only the physical objects actually being changed, space requirements for log files can be drastically reduced. One can save even more space by exploiting the fact that most access path structures consist of fully redundant information. For example, one can completely reconstruct a B*-tree from the record occurrences to which it refers. In itself, this type of reconstruction is certainly too expensive to become a standard method for crash recovery. But if only the modifications in the records are logged, after a crash the corresponding B* tree can be recovered consistently, provided that an appropriate write discipline has been observed for the pages containing the tree. This principle, stating that changed nodes must be written bottom up, is a special case of the "careful replacement" technique explained in detail by Verhofstadt [1978]. For our taxonomy it makes no difference whether the principle is applied or not.

Transition Logging on the Access Path Level. On the access path level, we are dealing with the entries of storage structures, but do not know how they are related to each other with regard to the objects of the database schema. This type of information is maintained on higher levels of the mapping hierarchy. If we look only at the physical entry representation (*physical transition logging*), state transition on this level means that a physical record, a table entry, etc. is *added* to, *deleted* from, or *modified* in a page. The arguments pertaining to these operations are the entries themselves, and so there is little difference between this and the previous approach. In the case of physical state logging on the access path level, we placed the physical address together with the entry representation. Here we place the operation code and object identifier with the same type of argument. Thus physical transition logging on this level does not provide anything essentially different.

We can also consider *logical* transition logging, attempting to exploit the syntax of the storage structures implemented on this level. The logical addition, a new record occurrence, for example, would include all the redundant table updates such as the record id index, the free space table, etc., each of which was explicitly logged with the physical schemes. Hence we again have a potential saving of log space. However, it is important to note that the logical transitions on this level generally affect *more than one page*. If they (or their inverse operators for UNDO) are to be applied during recovery, we must be sure that all affected pages have the same state in the materialized database. This is not the case with direct page allocation, and using the more expensive indirect schemes cannot be

TRANSACTION LOG

Figure 7. Logical transition logging as implemented in System R. (From Gray et al. [1981].)

justified by the comparatively few benefits yielded by logical transition logging on the access path level. Hence logical transition logging on this level can generally be ruled out, but will become more attractive on the next higher level.

Logical Logging on the Record-Oriented Level. At one level higher, it is possible to express the changes performed by the transaction program in a very compact manner by simply recording the update DML statements with their parameters. Even if a nonprocedural query language is being used above this level, its updates will be decomposed into updates of single records or tuples equivalent to the single-record updates of procedural DB languages. Thus logging on this level means that only the INSERT, UPDATE, and DELETE operations, together with their record ids and attribute values, are written to the log. The mapping process discerns which entries are affected, which pages must be modified, etc. Thus recovery is achieved by reexecuting some of the previously processed DML statements. For UNDO recovery, of course, the inverse DML statement must be executed, that is, a DELETE to compensate an INSERT and vice versa, and an UPDATE returned to the original values. These inverse DML statements must be generated automatically as part of the regular logging activity, and for this reason this approach is not viable for network-oriented DBMSs with information-bearing interrecord relations. In such cases, it can be extremely expensive to determine, for example, the inverse for a DELETE. Details can be found in Reuter [1981].

System R is a good example of a system with logical logging on the record-oriented level. All update operations performed on the tuples are represented by one generalized modification operator, which is not explicitly recorded. This operator changes a tuple identified by its tuple identifier (TID) from an old value to a new one, both of which are recorded. Inserting a tuple entails modifying its initial null value to the given value, and deleting a tuple entails the inverse transition. Hence the log contains the information shown in Figure 7.

Logical transition logging obviously requires a materialized database that is consistent up to Level 3; that is, it can only be combined with ATOMIC propagation schemes. Although the number of log data written are very small, recovery will be more expensive than that in other schemes, because it involves the reprocessing of some DML statements, although this can be done more cheaply than the original processing.

Table 3 is a summation of the properties of all logging techniques that we have described under two considerations: What is the cost of collecting the log data during normal processing? and, How expensive is recovery based on the respective type of log information? Of course, the entries in the table are only very rough qualitative estimations; for more detailed quantitative analysis see Reuter [1982].

Writing log information, no matter what type, is determined by two rules:

• UNDO information must be written to the log file *before* the corresponding updates are propagated to the materialized database. This has come to be known as the "write ahead log" (WAL) principle [Gray 1978].
• REDO information must be written to the temporary and the archive log file *before* EOT is acknowledged to the transaction program. Once this is done, the system must be able to ensure the transaction's durability.

We return to different facets of these rules in Section 3.4.

Table 3. Qualitative Comparison of Various Logging Techniques[a]

Logging technique	Level no.	Expenses during normal processing	Expenses for recovery operations
Physical state	2	High	Low
Physical transition	2	Medium	Low
Physical state	3	Low	Low
Logical transition	4	Very low	Medium

[a] Costs are basically measured in units of physical I/O operations. Recovery in this context means crash recovery.

3.3 Examples of Recovery Techniques

3.3.1 Optimization of Recovery Actions by Checkpoints

An appropriate combination of redundancy provided by log protocols and mapping techniques is basically all that we need for implementing transaction-oriented database recovery as described in Section 1. In real systems, however, there are a number of important refinements that reduce the amount of log data required and the costs of crash recovery. Figure 8 is a very general example of crash recovery. In the center, there is the temporary log containing UNDO and REDO information and special entries notifying the begin and end of a transaction (BOT and EOT, respectively). Below the temporary log, the transaction history preceding the crash is shown, and above it, recovery processing for global UNDO and partial REDO is related to the log entries. We have not assumed a specific propagation strategy.

There are two questions concerning the costs of crash recovery:

- In the case of the materialized DB being modified by incomplete transactions, to what extent does the log have to be processed for UNDO recovery?
- If the DBMS does not use a FORCE discipline, which part of the log has to processed for REDO recovery?

The first question can easily be answered: If we know that updates of incomplete transactions *can* have affected the materialized database (STEAL), we must scan the temporary log file back to the BOT entry of the *oldest* incomplete transaction to be sure that no invalid data are left in the system. The second question is not as simple. In Figure 8, REDO is started at a point that seems to be chosen arbitrarily. Why is there no REDO recovery for object A? In general, we can assume that in the case of a FORCE discipline modified pages will be written eventually because of buffer replacement. One might expect that only the contents of the most recently changed pages have to be redone—if the change was caused by a complete transaction. But look at a buffer activity record shown in Figure 9.

The situation depicted in Figure 9 is typical of many large database applications. Most of the modified pages will have been changed "recently," but there are a few hot spots like p_i, pages that are modified again and again, and, since they are *referenced* so frequently, have not been written from the buffer. After a while such pages will contain the updates of *many* complete transactions, and REDO recovery will therefore have to go back very far on the temporary log. This makes restart expensive. In general, the amount of log data to be processed for partial REDO will increase with the interval of time between two subsequent crashes. In other words, the higher the availability of the system, the more costly recovery will become. This is unacceptable for large, demanding applications.

For this reason additional measures are required for making restart costs independent of mean time between failure. Such provisions will be called *checkpoints*, and are defined as follows.

Generating a checkpoint means collecting information in a safe place, which has the effect of defining and limiting the amount of REDO recovery required after a crash.

Whether this information is stored in the log or elsewhere depends on which implementation technique is chosen; we give
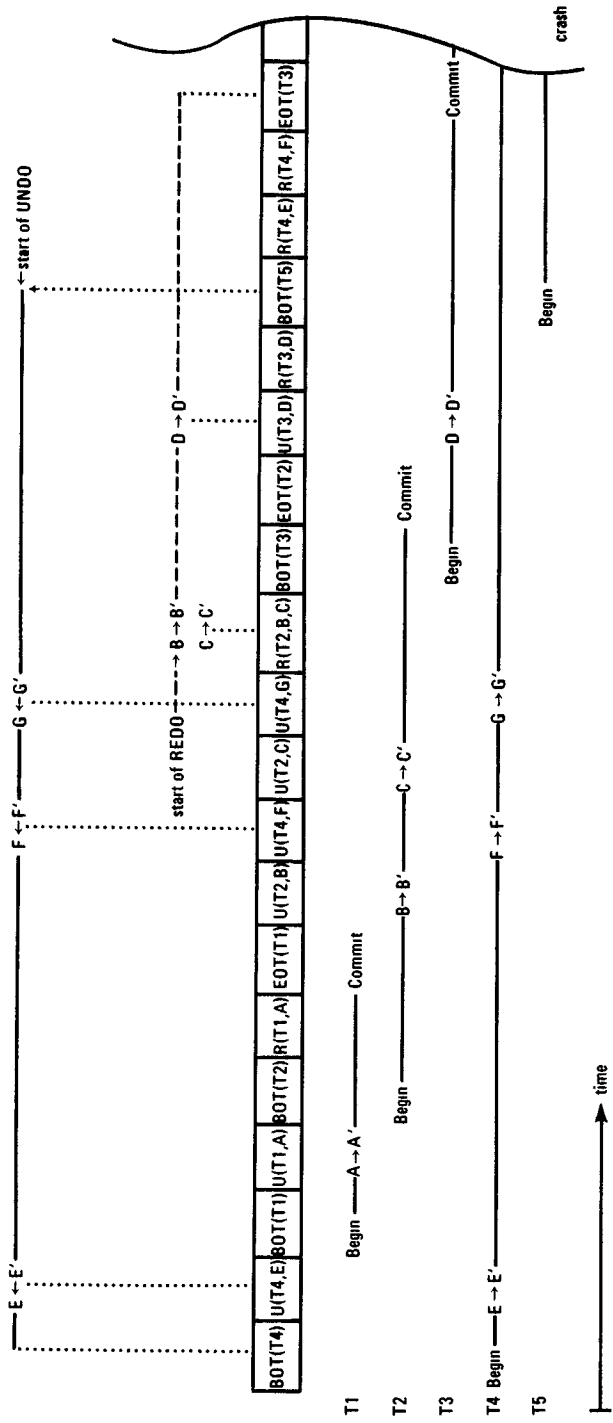
Figure 8. A crash recovery scenario.

Pages re-
siding in
buffer at
the mo-
ment of
Time   crash
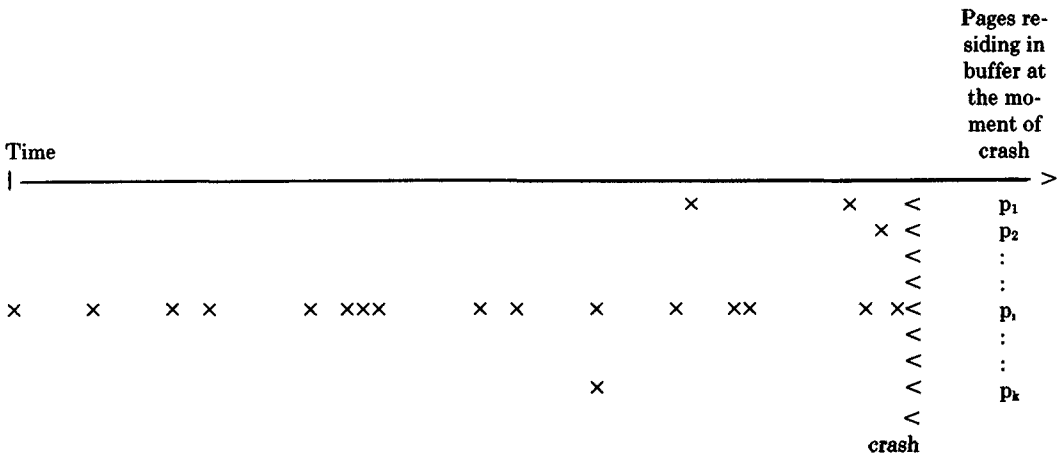

Figure 9.   Age of buffer page modifications (×, page modification).

some examples in this section. Checkpoint generation involves three steps [Gray 1978]:

- Write a BEGIN_CHECKPOINT record to the temporary log file.
- Write all checkpoint data to the log file and/or the database.
- Write an END_CHECKPOINT record to the temporary log file.

During restart, the BEGIN–END bracket is a clear indication as to whether a checkpoint was generated completely or interrupted by a system crash. Sometimes checkpointing is considered to be a means for restoring the whole database to some previous state. Our view, however, focuses on transaction recovery. Therefore to us a checkpoint is a technique for optimizing crash recovery rather than a definition of a distinguished state for recovery itself. In order to effectively constrain partial REDO, checkpoints must be generated at well-defined points in time. In the following sections, we shall introduce four separate criteria for determining when to start checkpoint activities.

3.3.2 Transaction-Oriented Checkpoints

As previously explained, a FORCE discipline will avoid partial REDO. All modified pages are propagated before an EOT record is written to the log, which makes the transaction durable. If this record is not found in the log after a crash, the transaction will
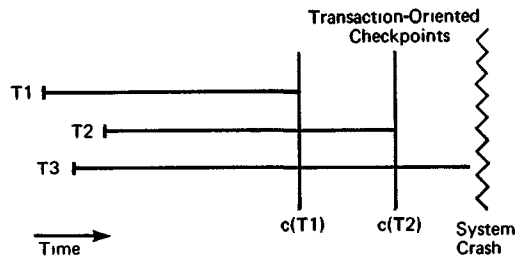

Figure 10.   Scenario for transaction-oriented checkpoints.

be considered incomplete and its effects will be undone. Hence the EOT record of each transaction can be interpreted as a BEGIN_CHECKPOINT and END_CHECKPOINT, since it agrees with our definition of a checkpoint in that it limits the scope of REDO. Figure 10 illustrates transaction-oriented checkpoints (TOC).

As can be seen in Figure 10, transaction-oriented checkpoints are implied by a FORCE discipline. The major drawback to this approach can be deduced from Figure 9. Hot spot pages like p_i will be propagated each time they are modified by a transaction even though they remain in the buffer for a long time. The reduction of recovery expenses with the use of transaction-oriented checkpoints is accomplished by imposing some overhead on normal processing. This is discussed in more detail in Section 3.5. The cost factor of unnecessary write operations performed by a FORCE discipline is highly relevant for very large
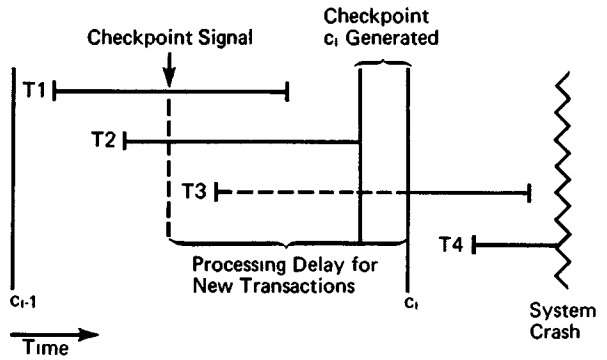
Figure 11. Scenario for transaction-consistent checkpoints.

database buffers. The longer a page remains in the buffer, the higher is the probability of multiple updates to the same page by different transactions. Thus for DBMSs supporting large applications, transaction-oriented checkpointing is not the proper choice.

3.3.3 Transaction-Consistent Checkpoints

The following transaction-consistent checkpoints (TCC) are *global* in that they save the work of all transactions that have modified the database. The first TCC, when successfully generated, creates a transaction-consistent database. It requires that all update activities on the database be quiescent. In other words, when the checkpoint generation is signaled by the recovery component, all incomplete update transactions are completed and new ones are not admitted. The checkpoint is actually generated when the last update is completed. After the END_CHECKPOINT record has been successfully written, normal operation is resumed. This is illustrated in Figure 11.

Checkpointing connotes propagating all modified buffer pages and writing a record to the log, which notifies the materialized database of a new transaction-consistent state, hence the name "transaction-consistent checkpoint" (TCC). By propagating all modified pages to the database, TCC establishes a point past which partial REDO will not operate. Since all modifications prior to the recent checkpoint are reflected in the database, REDO-log information need only be processed back to the youngest END_CHECKPOINT record found on the log. We shall see later on that the time between

two subsequent checkpoints can be adjusted to minimize overall recovery costs.

In Figure 11, T3 must be redone completely, whereas T4 must be rolled back. There is nothing to be done about T1 and T2, since their updates have been propagated by generating c_i. Favorable as that may sound, the TCC approach is quite unrealistic for large multiuser DBMSs, with the exception of one special case, which is discussed in Section 3.4. There are two reasons for this:

- Putting the system into a quiescent state until no update transaction is active may cause an intolerable delay for incoming transactions.
- Checkpoint costs will be high in the case of large buffers, where many changed pages will have accumulated. With a buffer of 6 megabytes and a substantial number of updates, propagating the modified pages will take about 10 seconds.

For small applications and single-user systems, TCC certainly is useful.

3.3.4 Action-Consistent Checkpoints

Each transaction is considered a sequence of elementary actions affecting the database. On the record-oriented level, these actions can be seen as DML statements. Action-consistent checkpoints (ACC) can be generated when no update *action* is being processed. Therefore signaling an ACC means putting the system into quiescence on the action level, which impedes operation here much less than on the transaction level. A scenario is shown in Figure 12.

The checkpoint itself is generated in the very same way as was described for the
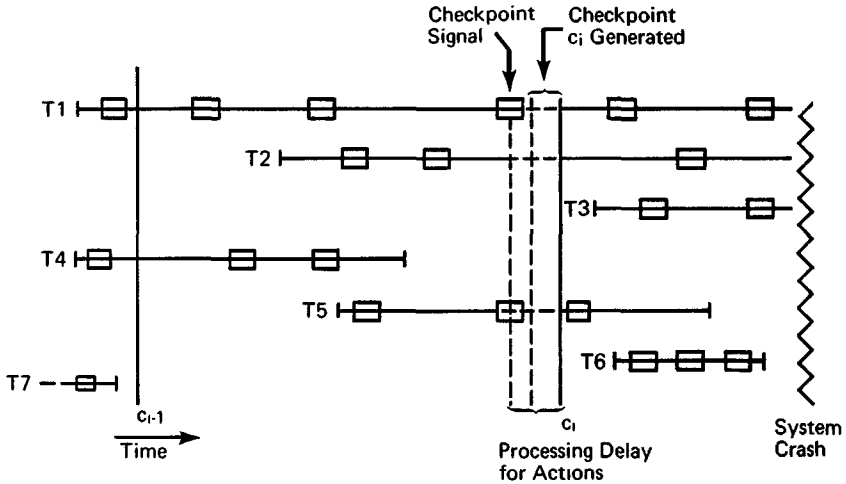
Figure 12. Scenario for action-consistent checkpoints.

TCC technique. In the case of ACC, however, the END_CHECKPOINT record indicates an action-consistent[4] rather than a transaction-consistent database. Obviously such a checkpoint imposes a limit on partial REDO. In contrast to TCC, it does not establish a boundary to global UNDO; however, it is not required by definition to do so. Recovery in the above scenario means global UNDO for T1, T2, and T3. REDO has to be performed for the last action of T5 and for all of T6. The changes of T4 and T7 are part of the materialized database because of checkpointing. So again, REDO-log information prior to the recent checkpoint is irrelevant for crash recovery. This scheme is much more realistic, since it does not cause long delays for incoming transactions. Costs of checkpointing, however, are still high when large buffers are used.

3.3.5 Fuzzy Checkpoints

In order to further reduce checkpoint costs, propagation activity at checkpoint time has to be avoided whenever possible. One way to do this is *indirect* checkpointing. Indirect checkpointing means that information about the buffer occupation is written to

the log file rather than the pages themselves. This can be done with two or three write operations, even with very large buffers, and helps to determine which pages containing committed data were actually in the buffer at the moment of a crash. However, if there are hot spot pages, their REDO information will have to be traced back very far on the temporary log. So, although indirect checkpointing does reduce the costs of partial REDO, this does not in general make partial REDO independent of mean time between failure. Note also that this method is only applicable with ¬ATOMIC propagation. In the case of ATOMIC schemes, propagation always takes effect at one well-defined moment, which is a checkpoint; pages that have only been written (not propagated) are lost after a crash. Since this checkpointing method is concerned only with the temporary log, leaving the database as it is, we call it "fuzzy." A description of a particular implementation of indirect, fuzzy checkpoints is given by Gray [1978].

The best of both worlds, low checkpoint costs with fixed limits to partial REDO, is achieved by another fuzzy scheme described by Lindsay *et al.* [1979]. This scheme combines ACC with indirect checkpointing: At checkpoint time the numbers of all pages (with an update indicator) currently in buffer are written to the log file. If there are no hot spot pages, nothing else
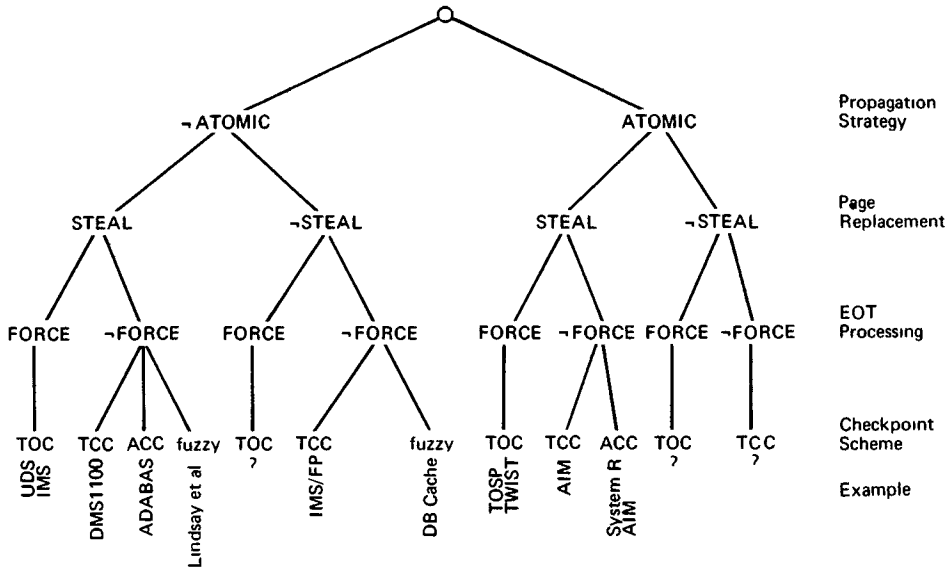
---

[4] This means that the materialized database reflects a state produced by complete actions only; that is, it is consistent up to Level 3 at the moment of checkpointing.

¬ATOMIC | ATOMIC | Propagation Strategy

STEAL | ¬STEAL | STEAL | ¬STEAL | Page Replacement

FORCE | ¬FORCE | FORCE | ¬FORCE | FORCE | ¬FORCE | FORCE | ¬FORCE | EOT Processing

TOC | TCC | ACC | fuzzy | TOC | TCC | fuzzy | TOC | TCC | ACC | TOC | TCC | Checkpoint Scheme

UDS IMS | DMS1100 | ADABAS | Lindsay et al | ? | IMS/FP | DB Cache | TOSP TWIST | AIM | System R AIM | ? | ? | Example

Figure 13. Classification scheme for recovery concepts.

is done. If, however, a modified page is found at two subsequent checkpoints without having been propagated, it will be propagated during checkpoint generation. Hence the scope of partial REDO is limited to two checkpoint intervals. Empiric studies show that the I/O activity for checkpointing is only about 3 percent of what is required with ACC [Reuter 1981]. This scheme can be given general applicability by adjusting the number of checkpoint intervals for modified pages in buffer.

Another fuzzy checkpoint approach has been proposed by Elhardt [1982]. Since a description of this technique, called database cache, would require more details than we can present in this paper, readers are referred to the literature.

3.4 Examples of Logging and Recovery Components

The introduction of various checkpoint schemes has completed our taxonomy. Database recovery techniques can now be classified as shown in Figure 13. In order to make the classification more vivid, we have added the names of a few existing DBMSs and implementation concepts to the corresponding entries.

In this section, we attempt to illustrate the functional principles of three different approaches found in well-known database systems. We particularly want to elaborate on the cooperation between mapping, logging, and recovery facilities, using a sample database constituting four pages, A, B, C, and D, which are modified by six transactions. What the transactions do is sketched in Figure 14. The indicated checkpoint c_1 is relevant only to those implementations actually applying checkpoint techniques. Prior to the beginning of Transaction 1 (T1), the DB pages were in the states A, B, C, and D, respectively.

3.4.1 Implementation Technique: ¬ATOMIC, STEAL, FORCE, TOC

An implementation technique involving the principles of ¬ATOMIC, STEAL, FORCE, and TOC can be found in many systems, for example, IMS [N.d.] and UDS [N.d.]. The temporary log file contains only UNDO data (owing to FORCE), whereas REDO information is written to the archive log. According to the write rules introduced in Section 3.2, we must be sure that UNDO logging has taken effect before a changed page is either replaced in the buffer or
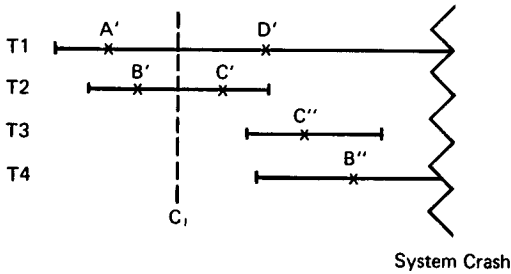
Figure 14. Transaction scenario for illustrating recovery techniques.

forced at EOT. Note that in ¬ATOMIC schemes EOT processing is interruptable by a crash.

In the scenario given in Figure 15, we need only consider T1 and T2; the rest is irrelevant to the example. According to the scenario, A′ has been replaced from the buffer, which triggered an UNDO entry to be written. Pages B′ and C′ remained in buffer as long as T2 was active. T2 reached its normal end before the crash, and so the following had to be done:

- Write UNDO information for B and C (in case the FORCE fails).
- Propagate B′ and C′.
- Write REDO information for B′ and C′ to the archive log file.
- Discard the UNDO entries for B and C.
- Write an EOT record to the log files and acknowledge EOT to the user.

Of course, there are some obvious optimizations as regards the UNDO data for pages that have not been replaced before EOT, but these are not our concern here. After the crash, the recovery component finds the database and the log files as shown in the scenario. The materialized database is inconsistent owing to ¬ATOMIC propagation, and must be made consistent by applying all UNDO information in reverse chronological order.

3.4.2 Implementation Technique: ¬ATOMIC, ¬STEAL, ¬FORCE, TCC

Applications with high transaction rates require large DB buffers to yield satisfactory performance. With sufficient buffer space, a ¬STEAL approach becomes feasible; that is, the materialized database will never contain updates of incomplete transactions. ¬FORCE is desirable for efficient EOT processing, as discussed previously (Section 3.3.2). The IMS/Fast Path in its "main storage database" version is a system designed with this implementation technique [IMS N.d.; Date 1981]. The ¬STEAL and ¬FORCE principles are generalized to the extent that there are no write operations to the database during normal processing. All updates are recorded to the log, and propagation is delayed until shutdown (or some other very infrequent checkpoint), which makes the system belong to the TCC class. Figure 16 illustrates the implications of this approach.

With ¬STEAL, there is no UNDO information on the temporary log. Accordingly, there are only committed pages in the materialized database. Each successful transaction writes REDO information during EOT processing. Assuming that the crash occurs as indicated in Figure 14, the materialized database is in the initial state, and, compared with the former current database, is old. Everything that has been done since start-up must therefore be applied to the database by processing the entire temporary log in chronological order. This, of course, can be very expensive, and hence the entire environment should be as stable as possible to minimize crashes. The benefits of this approach are extremely high transaction rates and short response times, since physical I/O during normal processing is reduced to a minimum.

The database cache, mentioned in Section 3.3, also tries to exploit the desirable properties of ¬STEAL and ¬FORCE, but, in addition, attempts to provide very fast crash recovery. This is attempted by implementing a checkpointing scheme of the "fuzzy" type.

3.4.3 Implementation Technique: ATOMIC, STEAL, ¬FORCE, ACC

ATOMIC propagation is not yet widely used in commercial database systems. This may result from the fact that indirect page mapping is more complicated and more expensive than the update-in-place technique. However, there is a well-known ex-

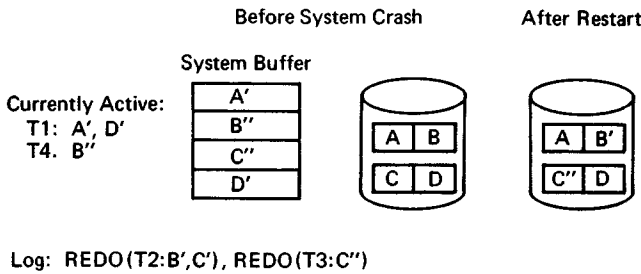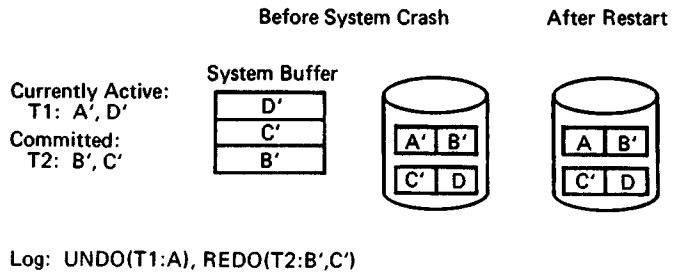**Figure 15.** Recovery scenario for ¬ATOMIC, STEAL, FORCE, TOC.

Before System Crash   After Restart

Currently Active:
T1: A', D'
Committed:
T2: B', C'

System Buffer
D'
C'
B'

Log: UNDO(T1:A), REDO(T2:B',C')

Before System Crash   After Restart

Currently Active:
T1: A', D'
T4. B''

System Buffer
A'
B''
C''
D'

Figure 16. Recovery scenario for ¬ATOMIC, ¬STEAL, ¬FORCE, TCC.

Log: REDO(T2:B',C'), REDO(T3:C'')

ample of this type of implementation, based on the shadow-page mechanism in System R. This system uses action-consistent checkpointing for update propagation, and hence comes up with a consistent materialized database after a crash. More specifically, the materialized database will be consistent up to Level 4 of the mapping hierarchy and reflect the state of the most recent checkpoint; everything occurring after the most recent checkpoint will have disappeared. As discussed in Section 3.2, with an action-consistent database one can use logical transition logging based on DML statements, which System R does. Note that in the case of ATOMIC propagation the WAL principle is bound to the propagation, that is, to the checkpoints. In other words, modified pages can be written, but not propagated, without having written an UNDO log. If the modified pages pertain to incomplete transactions, the UNDO information must be on the temporary log before the pages are propagated. The same is true for STEAL: Not only can dirty pages be written; in the case of System R they can also be propagated. Consider the scenario in Figure 17.

T1 and T2 were both incomplete at checkpoint. Since their updates (A' and B') have been propagated, UNDO information must be written to the temporary log. In System R, this is done with logical transi-

tions, as described in Section 3.2. EOT processing of T2 and T3 includes writing REDO information to the log, again using logical transitions. When the system crashes, the current database is in the state depicted in Figure 17; at restart the materialized database will reflect the most recent checkpoint state. Crash recovery involves the following actions:

- UNDO the modification of A'. Owing to the STEAL policy in System R, incomplete transactions can span several checkpoints. Global UNDO must be applied to all changes of failed transactions prior to the recent checkpoint.
- REDO the last action of T2 (modification of C') and the whole transaction T3 (modification of C''). Although they are committed, the corresponding page states are not yet reflected in the materialized database.
- Nothing has to be done with D' since this has not yet become part of the materialized database. The same is true of T4. Since it was not present when c_i was generated, it has had no effect on the materialized database.

3.5 Evaluation of Logging and Recovery Concepts

Combining all possibilities of propagating, buffer handling, and checkpointing, and
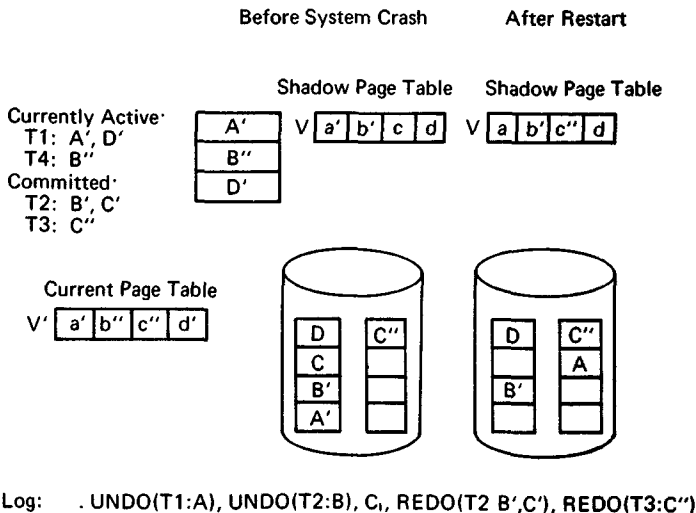
Before System Crash   After Restart

Figure 17. Recovery scenario for ATOMIC, STEAL, ¬FORCE, ACC.

Log: . UNDO(T1:A), UNDO(T2:B), C_i, REDO(T2 B',C'), REDO(T3:C'')

Table 4. Evaluation of Logging and Recovery Techniques Based on the Introduced Taxonomy

propagation strategy	¬ATOMIC							ATOMIC				
buffer replacement	STEAL				¬STEAL			STEAL			¬STEAL	
EOT processing	FORCE	¬FORCE			FORCE	¬FORCE		FORCE	¬FORCE		FORCE	¬FORCE
checkpoint type	TOC	TCC	ACC	FUZZY	TOC	TCC	FUZZY	TOC	TCC	ACC	TOC	TCC
materialized DB state after system failure	DC	DC	DC	DC	DC	DC	DC	TC	TC	AC	TC	TC
cost of transaction UNDO	+	+	+	+	--	--	--	-	-	+	--	--
cost of partial REDO at restart	--	-	-	+	--	-	-	--	-	-	--	-
cost of global UNDO at restart	+	+	+	+	--	--	--	--	--	+	--	--
overhead during normal processing	--	--	--	--	--	--	--	+	+	+	+	+
frequency of check-points	+	-	-	-	+	-	-	+	-	-	+	-
checkpoint cost	+	++	++	-	+	++	+	+	++	++	+	++

Notes:
Abbreviations: DC, device consistent (chaotic); AC, action consistent; TC, Transaction consistent.
Evaluation symbols: ——, very low; —, low; +, high; ++, very high.

considering the overall properties of each scheme that we have discussed, we can derive the evaluation given in Table 4.

Table 4 can be seen as a compact summary of what we have discussed up to this point. Combinations leading to inherent contradictions have been suppressed (e.g., ¬STEAL does not allow for ACC). By referring the information in Table 4 to Figure 13; one can see how existing DBMSs are rated in this qualitative comparison.

Some criteria of our taxonomy divide the world of DB recovery into clearly distinct areas:

- ATOMIC propagation achieves an action- or transaction-consistent materialized database in the event of a crash. Physical as well as logical logging techniques are therefore applicable. The benefits of this property are offset by increased overhead during normal processing caused by the redundancy required for indirect page mapping. On the other hand, recovery can be cheap when ATOMIC propagation is combined with TOC schemes.
- ¬ATOMIC propagation generally results in a chaotic materialized database in the event of a crash, which makes physical logging mandatory. There is almost no overhead during normal processing, but without appropriate checkpoint schemes, recovery will more expensive.
- All transaction-oriented and transaction-consistent schemes cause high checkpoint costs. This problem is emphasized in transaction-oriented schemes by a relatively high checkpoint frequency.

It is, in general, important when deciding which implementation techniques to choose for database recovery to carefully consider whether optimizations of crash recovery put additional burdens on normal processing. If this is the case, it will certainly not pay off, since crash recovery, it is hoped, will be a rare event. Recovery components should be designed with minimal overhead for normal processing, provided that there is fixed limit to the costs of crash recovery.

This consideration rules out schemes of the ATOMIC, FORCE, TOC type, which can be implemented and look very appealing at first sight. According to the classification, the materialized database will always be in the most recent transaction-consistent state in implementations of these schemes. Incomplete transactions have not affected the materialized database, and successful transactions have propagated indivisibly during EOT processing. However appealing the schemes may be in terms of crash recovery, the overhead during normal processing is too high to justify their use [Haerder and Reuter 1979; Reuter 1980].

There are, of course, other factors influencing the performance of a logging and recovery component: The granule of logging (pages or entries), the frequency of checkpoints (it depends on the transaction load), etc. are important. Logging is also tied to concurrency control in that the granule of logging determined the granule of locking. If page logging is applied, DBMS must not use smaller granules of locking than pages. However, a detailed discussion of these aspects is beyond the scope of this paper; detailed analyses can be found in Chandy et al. [1975] and Reuter [1982].

4. ARCHIVE RECOVERY

Throughout this paper we have focused on crash recovery, but in general there are two types of DB recovery, as is shown in Figure 18. The first path represents the standard crash recovery, depending on the physical (and the materialized) database as well as on the temporary log. If one of these is lost or corrupted because of hardware or software failure, the second path, archive recovery, must be tried. This presupposes that the components involved have independent failure modes, for example, if temporary and archive logs are kept on different devices. The global scenario for archive recovery is shown in Figure 19; it illustrates that the component "archive copy" actually depends on some dynamically modified subcomponents. These subcomponents create new archive copies and update existing ones. The following is a brief sketch of some problems associated with this.

Creating an archive copy, that is, copying the on-line version of the database, is a very expensive process. If the copy is to be consistent, update operation on the database has to be interrupted for a long time, which is unacceptable in many applications. Archive recovery is likely to be rare, and an archive copy should not be created too frequently, both because of cost and because there is a chance that it will never be used. On the other hand, if the archive copy is very old, recovery starting from such a copy will have to redo too much work and will take too long. There are two methods to cope with this. First, the database can be copied on the fly, that is, without inter-
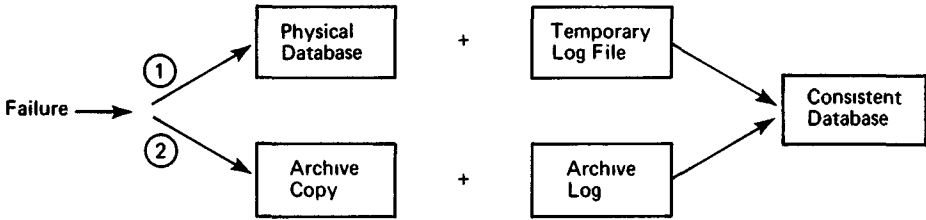
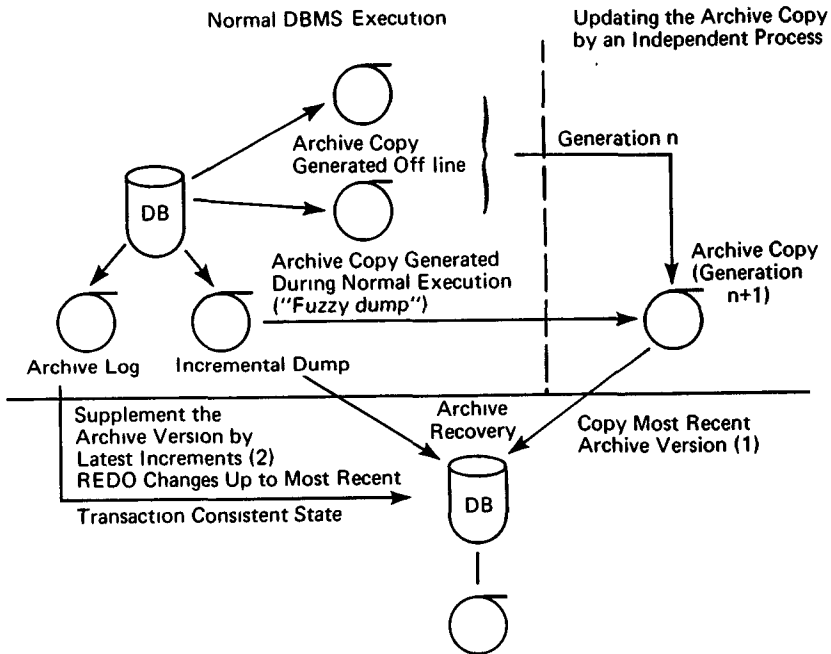**Figure 18.** Two ways of DB recovery and the components involved.

Figure 19. Scenario for archive recovery (global REDO).

rupting processing, in parallel with normal processing. This will create an inconsistent copy, a so-called "fuzzy dump."

The other possibility is to write only the changed pages to an incremental dump, since a new copy will be different from an old one only with respect to these pages. Either type of dump can be used to create a new, more up-to-date copy from the previous one. This is done by a separate off-line process with respect to the database and therefore does not affect DB operation. In the case of DB applications running 24 hours per day, this type of separate process is the only possible way to maintain archive recovery data. As shown in Figure 19, ar-

chive recovery in such an environment requires the most recent archive copy, the latest incremental modifications to it (if there are any), and the archive log. When recovering the database itself, there is little additional cost in creating an identical new archive copy in parallel.

There is still another problem hidden in this scenario: Since archive copies are needed very infrequently, they may be susceptible to magnetic decay. For this reason several generations of the archive copy are usually kept. If the most recent one does not work, its predecessor can be tried, and so on. This leads to the consequences illustrated in Figure 20.
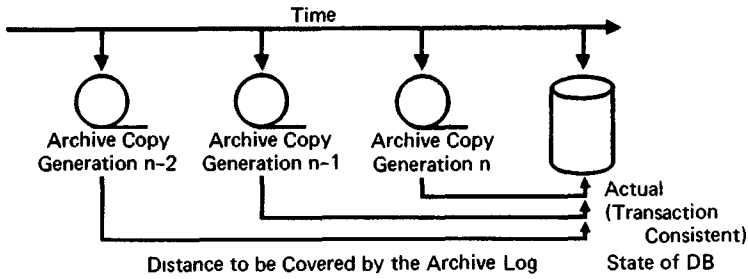
Figure 20. Consequences of multigeneration archive copies.
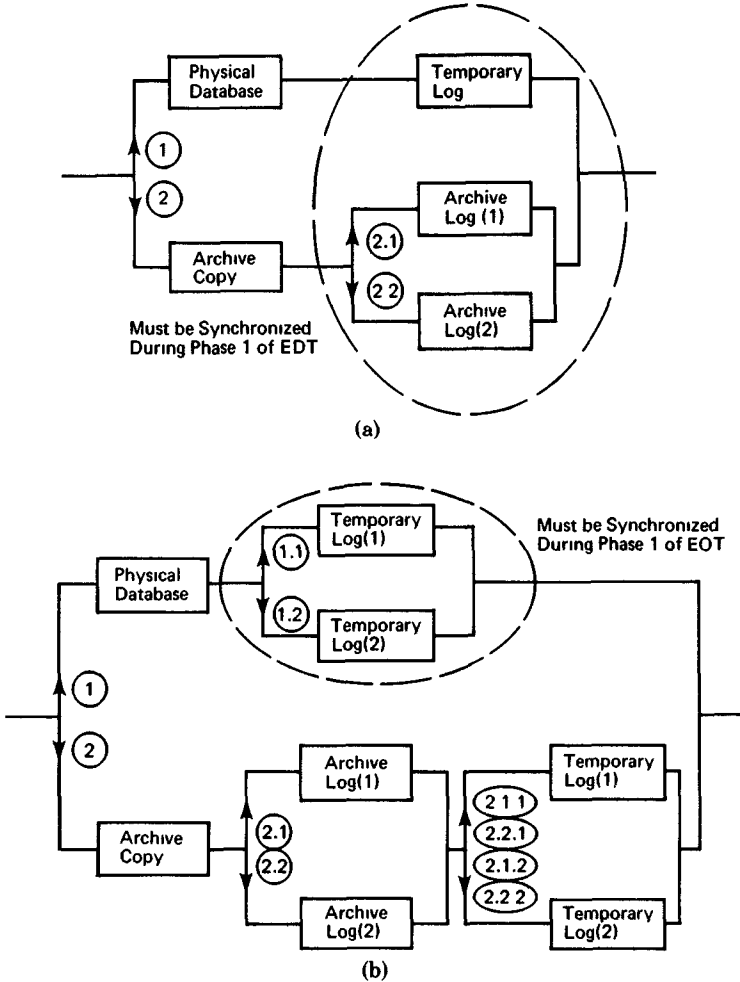

Figure 21. Two possibilities for duplicating the archive log.

We must anticipate the case of starting archive recovery from the *oldest* generation, and hence the archive log must span the whole distance back to this point in time. That makes the log susceptible to magnetic decay, as well, but in this case generations will not help; rather we have to duplicate the entire archive log file. Without taking

storage costs into account, this has severe impact on normal DB processing, as is shown in Figure 21.

Figure 21a shows the straightforward solution: two archive log files that are kept on different devices. If this scheme is to work, all three log files must be in the same state at any point in time. In other words, writing to these files must be synchronized *at each EOT*. This adds substantial costs to normal processing and particularly affects transaction response times. The solution in Figure 21b assumes that *all log information* is written only to the temporary log during normal processing. An independent process that runs asynchronously then copies the REDO data to the archive log. Hence archive recovery finds most of the log entries in the archive log, but the temporary log is required for the most recent information. In such an environment, temporary and archive logs are no longer independent from a recovery perspective, and so we must make the temporary log very reliable by duplicating it. The resulting scenario looks much more complicated than the first one, but in fact the only additional costs are those for temporary log storage—which are usually small. The advantage here is that only two files have to be synchronized during EOT, and moreover—as numerical analysis shows—this environment is more reliable than the first one by a factor of 2.

These arguments do not, of course, exhaust the problem of archive recovery. Applications demanding very high availability and fast recovery from a media failure will use additional measures such as duplexing the whole database and all the hardware (e.g., see TANDEM [N.d.]). This aspect of database recovery does not add anything conceptually to the recovery taxonomy established in this paper.

5. CONCLUSION

We have presented a taxonomy for classifying the implementation techniques for database recovery. It is based on four criteria:

Propagation. We have shown that update propagation should be carefully distinguished from the write operation. The ATOMIC/¬ATOMIC dichotomy defines two different methods of handling low-level updates of the database, and also gives rise to different views of the database, both the materialized and the physical database. This proves to be useful in defining different crash states of a database.

Buffer Handling. We have shown that interfering with buffer replacement can support UNDO recovery. The STEAL/¬STEAL criterion deals with this concept.

EOT Processing. By distinguishing FORCE policies from ¬FORCE policies we can distinguish whether successful transactions will have to be redone after a crash. It can also be shown that this criterion heavily influences the DBMS performance during normal operation.

Checkpointing. Checkpoints have been introduced as a means for limiting the costs of partial REDO during crash recovery. They can be classified with regard to the events triggering checkpoint generation and the number of data written at a checkpoint. We have shown that each class has some particular performance characteristics.

Some existing DBMSs and implementation concepts have been classified and described according to the taxonomy. Since the criteria are relatively simple, each system can easily be assigned to the appropriate node of the classification tree. This classification is more than an ordering scheme for concepts: Once the parameters of a system are known, it is possible to draw important conclusions as to the behavior and performance of the recovery component.

REFERENCES

ASTRAHAN, M. M., BLASGEN, M. W., CHAMBERLIN, D. D., GRAY, J. N., KING, W. F., LINDSAY, B. G.,

LORIE, R., MEHL, J. W., PRICE, T. G., PUTZOLU, F., SELINGER, P. G., SCHKOLNICK, M., SLUTZ, D. R., TRAIGER, I. L., WADE, B. W., AND YOST, R. A. 1981. History and evaluation of System R. *Commun. ACM 24*, 10 (Oct.), 632–646.

BERNSTEIN, P. A., AND GOODMAN, N. 1981. Concurrency control in distributed database systems. *ACM Comput. Surv. 13*, 2 (June), 185–221.

BJORK, L. A. 1973. Recovery scenario for a DB/DC system. In *Proceedings of the ACM 73 National Conference* (Atlanta, Ga., Aug. 27–29). ACM, New York, pp. 142–146.

CHAMBERLIN, D. D. 1980. A summary of user experience with the SQL data sublanguage. In *Proceedings of the International Conference on Databases* (Aberdeen, Scotland, July), S. M. Deen and P. Hammersley, Eds. Heyden, London, pp. 181–203.

CHANDY, K. M., BROWN, J. C., DISSLEY, C. W., AND UHRIG, W. R. 1975. Analytic models for rollback and recovery strategies in data base systems. *IEEE Trans Softw Eng. SE-1*, 1 (Mar.), 100–110.

CHEN, T. C. 1978. Computer technology and the database user. In *Proceedings of the 4th International Conference on Very Large Database Systems* (Berlin, Oct.). IEEE, New York, pp. 72–86.

CODASYL 1973. *CODASYL DDL Journal of Development* June Report. Available from IFIP Administrative Data Processing Group, 40 Paulus Potterstraat, Amsterdam.

CODASYL 1978. CODASYL: Report of the Data Description Language Committee. *Inf. Syst. 3*, 4, 247–320.

CODD, E. F. 1982. Relational database: A practical foundation for productivity. *Commun. ACM 25*, 2 (Feb.), 109–117.

DATE, C. J. 1981. *An Introduction to Database Systems*, 3rd ed. Addison-Wesley, Reading, Mass.

DAVIES, C. T. 1973. Recovery semantics for a DB/DC System. In *Proceedings of the ACM 73 National Conference,* (Atlanta, Ga., Aug. 27–29). ACM, New York, pp. 136–141.

DAVIES, C. T. 1978. Data processing spheres of control. *IBM Syst. J. 17*, 2, 179–198.

EFFELSBERG, W., HAERDER, T., REUTER, A., AND SCHULZE-BOHL, J. 1981. Performance measurement in database systems—Modeling, interpretation and evaluation. In *Informatik Fachberichte 41*. Springer-Verlag, Berlin, pp. 279–293 (in German).

ELHARDT, K. 1982. The database cache—Principles of operation. Ph.D. dissertation, Technical University of Munich, Munich, West Germany (in German).

ESWARAN, K. P., GRAY, J. N., LORIE, R. A., AND TRAIGER, I. L. 1976. The notions of consistency and predicate locks in a database system. *Commun. ACM 19*, 11 (Nov.), 624–633.

GRAY, J. 1978. Notes on data base operating systems. In *Lecture Notes on Computer Science*, vol. 60, R. Bayer, R. N. Graham, and G. Seegmueller, Eds. Springer-Verlag, New York.

GRAY, J. 1981. The transaction concept: Virtues and limitations. In *Proceedings of the 7th International Conference on Very Large Database Systems* (Cannes, France, Sept. 9–11). ACM, New York, pp. 144–154.

GRAY, J., LORIE, R., PUTZOLU, F., AND TRAIGER, I. L. 1976. Granularity of locks and degrees of consistency in a large shared data base. In *Modeling in Data Base Management Systems*. Elsevier North-Holland, New York, pp. 365–394.

GRAY, J., McJONES, P., BLASGEN, M., LINDSAY, B., LORIE, R., PRICE, T., PUTZOLU, F., AND TRAIGER, I. L. 1981. The recovery manager of the System R database manager. *ACM Comput. Surv. 13*, 2 (June), 223–242.

HAERDER, T., AND REUTER, A. 1979. Optimization of logging and recovery in a database system. In *Database Architecture*, G. Bracchi, Ed. Elsevier North-Holland, New York, pp. 151–168.

HAERDER, T., AND REUTER, A. 1983. Concepts for implementing a centralized database management system. In *Proceedings of the International Computing Symposium* (Invited Paper) (Nuernberg, W. Germany, Apr.), H. J. Schneider, Ed. German Chapter of ACM, B. G. Teubner, Stuttgart, pp. 28–60.

IMS/VS-DB N.d. IMS/VS-DB Primer, IBM World Trade Center, Palo Alto, July 1976.

KOHLER, W. H. 1981. A survey of techniques for synchronization and recovery in decentralized computer systems. *ACM Comput. Surv. 13*, 2 (June), 149–183.

LAMPSON, B. W., AND STURGIS, H. E. 1979. Crash recovery in a distributed data storage system. XEROX Res. Rep. Palo Alto, Calif. Submitted for publication.

LINDSAY, B. G., SELINGER, P. G., GALTIERI, C., GRAY, J. N., LORIE, R., PRICE, T. G., PUTZOLU, F., TRAIGER, I. L., AND WADE, B. W. 1979. Notes on distributed databases. IBM Res. Rep. RJ 2571, San Jose, Calif.

LORIE, R. A. 1977. Physical integrity in a large segmented database. *ACM Trans. Database Sys. 2*, 1 (Mar.), 91–104.

REUTER, A. 1980. A fast transaction-oriented logging scheme for UNDO-recovery. *IEEE Trans. Softw. Eng. SE-6* (July), 348–356.

REUTER, A. 1981. *Recovery in Database Systems*. Carl Hanser Verlag, Munich (in German).

REUTER, A. 1982. Performance Analysis of Recovery Techniques, Res. Rep., Computer Science De-

partment, Univ. of Kaiserslautern, 1982. To be published.

SENKO, M. E., ALTMAN, E. B., ASTRAHAN, M. M., AND FEHBER, P. L. 1973. Data structures and accessing in data base systems. *IBM Syst. J. 12*, 1 (Jan), 30–93.

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

SMITH, D. D. P., AND SMITH, J. M. 1979. Relational database machines. *IEEE Comput. 12*, 3 28–38.

STONEBRAKER, M. 1980. Retrospection on a database system. *ACM Trans. Database Syst. 5*, 2 (June), 225–240.

STONEBRAKER, M., WONG, E., KREPS, P., AND HELD, G. 1976. The design and implementation of INGRES. *ACM Trans. Database Syst. 1*, 3 (Sept.), 189–222.

STURGIS, H., MITCHELL, J., AND ISRAEL, J. 1980. Issues in the design and use of a distributed file system. *ACM Oper. Syst. Rev. 14*, 3 (July), 55–69.

TANDEM. N.d. TANDEM 16, ENSCRIBE Data Base Record Manager, Programming Manual, TANDEM Computer Inc., Cupertino.

UDS, N.d. UDS, Universal Data Base Management System, UDS-V2 Reference Manual Package, Siemens AG, Munich, West Germany.

VERHOFSTADT, J. M. 1978. Recovery techniques for database systems. *ACM Comput. Surv. 10*, 2 (June), 167–195.