An Introduction to Database Systems

2nd Edition

Bipin C. DESAI

Concordia University
Montreal

BytePress

Limit of Liability/Disclaimer of Warranty:

The authors and the publishers have taken care to prepare this book. However, there is no warranty of the accuracy, completeness or presentation of the latest version/generation of any system discussed in this book. The reader must be aware of the fact that software systems often have multiple bugs and are not well thought out, and are usually suitable for limited situations and/or data combinations. Hence the user must be responsible for the appropriate application of any technique and use of any software or code examples.

Furthermore, there is no assurance whatsoever of the possible usefulness or commercialization of any programs, scripts and examples given in this book.

Any references given are based on their existence at the time of writing and the authors and the publishers do not endorse them or imply any usefulness of the information found therein. The reader must be aware that any web site cited may change, disappear or change their terms of service.

Published by: Electronic Publishing BytePress.com Inc.
Hardcopy - ISBN: 978-1-988392-15-8
Electronic - ISBN: 978-1-988392-08-0

11 Recovery

A computer system is an electro-mechanical device subject to failures of various types. The reliability problem of the database system is linked to the reliability of the computer system on which it runs. In this chapter we will discuss the recovery of the data contained in a database system following failures of various types and present the different approaches to database recovery. The types of failures that the computer system is likely to be subjected to include failures of components or subsystems, software failures, power outages, accidents, unforeseen situations, and natural or man-made disasters. Database recovery techniques are methods of making the database fault-tolerant. The aim of the recovery scheme is to allow database operations to be resumed after a failure, with minimum loss of information, at an economically justifiable cost. We concentrate on the recovery of centralized database systems in this chapter; the recovery issues in a distributed system are presented in chapter 13.

11.1 Reliability

A system is considered **reliable** if it functions as per its specifications and produces a correct set of output values for a given set of input values. For a computer system, reliable operation is said to be attained when all components of the system work according to specifications. This in turn requires that the system which consists of both software and hardware (in which we include firmware) is working correctly. The **failure** of a system is said to occur when the system does not function according to its specifications and fails to deliver the service for which it was intended. An **error** in the system occurs when a component of the system assumes a state that is not desirable: the fact that the state is undesirable is a subjective judgement. The component in question is said to be in an erroneous state and further use of the component will lead to a failure which cannot be attributed to any other factor. A **fault** is said to be detected when either an error is propagated from one component to another or the failure of the component is observed. Sometimes it may not be possible to attribute a fault to a specific cause. Furthermore, errors, such as logical errors in a program, are latent as long as they do not manifest themselves as faults at some unspecified time. A fault is, in effect, the identified or assumed cause of an error. An error, if it is not propagated or perceived by another component of a system or by an user, may not be considered as a failure.

Consider a bank teller who requests the balance of an account from the database system. If there is an unrecoverable parity error in trying to read the specific information, then the system would return the response to the teller that it was unable to retrieve the required information; furthermore, the system will make a report of this error and its cause as being a parity error to a system error log. The cause of the parity error could be a fault in the disk drive or memory location containing the required information: or the problem could be traced to poor interconnection or noise on the communication lines; finally one cannot rule out the fact that the parity checking unit itself may be defective.

For a database system (or for that matter, any other system) works correctly, we need correct data, correct algorithms to manipulate the data, correct programs that implement these algorithms, and of course a computer system that functions correctly. Any source of errors in each of these components has to be identified, and a method of correcting and recovering from these errors has to be designed in the system. To ensure that data is correct, validation checks have to be incorporated for data entry functions.

For example, if the age of an employee is entered as being too low or too high, then the validation routine should ask for a confirmation of the data that was entered.

Fault detection schemes of appropriate types have to be built into a reliable system. These will detect any errors that may manifest themselves. In addition, a reliable system has built into it, appropriate recovery schemes that will correct the errors that have been detected, or eliminate a portion of the permanently failed system; such elimination, however, may mean that the system may not be available until it is repaired.

A **fault-tolerant system**, in addition to the fault detection scheme, has redundant components and subsystems built into it. On detection of a fault, these redundant components are used to replace the faulty components. Such replacement makes it possible to continue to have the system available without any interruption of service, albeit, at a reduced level of performance and reliability.

We will not consider the aspects of correct algorithms or correct implementation of these algorithms in this text. However, we stress their paramount importance in the correct functioning of any system including a database system.

Another aspect that has to be considered in database application is that of data consistency. Having correct data is important, however, the data must be consistent. This requires that there be checks in the database system to ensure that any redundant data is consistent. For example, if the age of an employee is entered in the database, it must be consistent with the employee's date of birth and the current date.

Let us now try to informally define the concept of reliability of a system. **Reliability** is a measure that is used to indicate how successful a system is in providing the service it was intended for. Reliability is an important consideration in all systems that are designed for critical operations. It is taken into consideration during all stages of computer system design and implementation. To take into account the fact that physical devices have an inherent failure rate, these systems include various mechanisms which can detect errors and correct many of these errors. There are a number of measures used to define the reliability of a system: these include the ***mean time between failures*** (**MTBF**), the ***mean time to repair*** (**MTTR**), and the **system availability** which is the fraction of time that a system performs according to its specifications.

There are two basic methods of increasing the reliability of a system: the first method uses fault avoidance and the second method tolerates faults and corrects these faults. In the fault avoidance method, reliability is achieved by using reliable components and using careful assembling techniques with comprehensive testing at each stage of the design and assembly, to eliminate all sources of hardware and software errors. In the fault tolerance approach, the system incorporates protective redundancies which can cater to faults occurring within the system and its components. These redundancies allow the system to perform according to its specifications (or within an acceptable level of degradation from these specifications). However, the use of redundancy, in components and subsystems to make a system fault tolerant, increases the number of components. A greater number of components in a system will decrease its reliability unless the components are modular and the redundant components do not get in the way of operation of the system's normal components. The modular construction effectively reduces the complexity of the system and the redundant components come into play only in the case of an error.

Memory systems can have a simple parity check bit which can detect a single bit error correctly, but multiple bit errors can go undetected (or detected incorrectly as a single bit error). However, if the costs are justified, then memory systems are made fault-tolerant by additional parity bits to detect and correct

errors in one or more bits. The degree to which such detection and correction schemes are used depends on the expected number of errors and the costs that can be economically justified.

Absolute reliability is hard to achieve at an economically acceptable cost (or at any cost), and, hence, systems are designed with a level of reliability that is compatible with the use of the system and is economically justifiable.

In database systems, reliability of the system is achieved by using redundancy of data including control data. In addition, failures are tolerated by using additional redundant data which can be used in recovery operations to return the database to an usable state, after the occurrence of a failure.

11.1.1 Types of Failure

Hardware Failure

The failure that can occur in the hardware could be attributed to one of the following sources: design errors, inadequate quality control, overloading, and wear-out.

Design Errors: These could include a design that did not meet the required specifications of performance and or reliability; the use of components that are of poor quality or not sufficient capacity; poor error detection and correction schemes; failure to take into account the errors that can occur in the error detection and correction subsystems.

Poor Quality Control (during Fabrication): This could include poor connections, defective subsystems and electrical and mechanical mis-alignments.

Over-utilization and overloading: Using a component or subsystem beyond its capacity. This could be a design error or utilization error where mismatching sub-components may be used, or due to unforeseen circumstances a system is simply used beyond its capacity.

Wear-out: The system, especially its mechanical parts, tend to wear with usage causing the system to divert from its design performance. Solid state electrical parts do not wear-out, but insulation on wire could undergo chemical changes with age and crack leading to eventual failure.

Software Failure

The source of errors that can lead to a software failure is similar to those that lead to hardware failure, the only exception being wear-out. We discuss these below.

Design errors: Not all possible situations could have been accounted for in the design process. This is particularly so in software design where it is hard to foresee all possible modes of operations, including the combinations and the sequence of usage of various components of a software system. However, the design should allow for the most serious types of errors to be detected and appropriate corrective action to be incorporated, for servicing and recovering from such errors. In situations which could result in loss of life or property, the design must be fail-safe. An alternate approach to design in such a situation is to assign multiple design teams for the same project and an independent verification team to validate the design.

Poor Quality Control: This could include undetected errors in entering the program code. Incompatibility of various modules and conflict of conventions between versions of the Operating System are other possible causes of failure in software.

Over-utilization and overloading: Here a system designed to handle a certain load may be swamped, when loading on it is exceeded. Buffers and stacks may overrun their boundaries, or be shared erroneously.

Wear-out: There are no known errors due to wearout of software: software does not wearout. However, the usefulness of a software system may become obsolete due to the introduction of new versions with additional features.

Storage Medium Failure

Storage media can be classified as being of the following types: the **volatile** type, the **nonvolatile** type, of the **permanent** or **stable** type.

Volatile Storage: An example of this type of storage is the semiconductor memory requiring an uninterruptable power source for correct operation. A volatile storage failure can occur due to the spontaneous shutdown of the computer system, sometimes referred to as a **system crash**. The cause of the shutdown could be due to a failure in the power supply unit, or a loss of power. A system crash will result in the loss of the information stored in the volatile storage medium. One method of avoiding loss of data due to power outages is to provide for uninterruptable power source (using batteries and or standby electrical generators). Another source of data loss from volatile storage can be due to parity errors in more bits than could be corrected by the parity checking unit, and such errors will cause partial loss of data.

Nonvolatile storage: Examples of this type of storage are magnetic tape and magnetic disk systems. These types of storage devices do not require power for maintaining the stored information. A power failure or system shutdown will not result in the loss of information stored on such devices. However, nonvolatile storage devices such as magnetic disks (**Hard Disk Drive - HDD**)can have a mechanical failure in the form of a **read/write head crash** (i.e., the read/write head coming in contact with the recording surface instead of being a small distance from it), which could result in some loss of information. It is vital that failures, which cause loss of ordinary data, should not also cause the loss of the redundant data that is to be used for recovery of the ordinary data. Thus, a head crash must not cause loss of both the ordinary data and the recovery data. One method of avoiding this double loss is to store the recovery data on separate storage devices. To avoid the loss of recovery data (primary recovery data), one can provide for a further set of recovery data, (secondary recovery data) and so on. However, this multiple level of redundancy can only be carried to an economically justifiable level.

With the introduction of **Solid State Drive - SSD** to this category of storage the disadvantages of seek time and latency of HDD has been removed. The speed of transfer is also much higher with SSD, Since there are no mechanical components to wear out or cause head crashes the reliability is higher. However, since the storage e requires ‘flash’ storage and since the life time of SSD is dependent on the number of write cycles for the type of flash storage used. SSD tend to fail after the number of write cycle exceeds this limit.[1]

---

1 The author has had the misfortune of having at least three SSD fail but no HDD in the last decade in servers etc. used in the labs.

Permanent or **Stable storage**: Permanency of storage, in view of the possibility of failure of the storage medium, is achieved by redundancy. Thus, instead of having a single copy of the data on a nonvolatile storage medium, multiple copies of the data are stored. Each such copy is made on a separate nonvolatile storage device. Since these independent storage devices have independent failure modes, it is assumed that at least one of these multiple copies will survive any failure and be usable. The amount and type of data stored in stable storage depends on the recovery scheme used in the particular DBMS. The status of the database at a given point in time is called **archive database** and such archive data is usually stored in stable storage. Recovery data which would be used to recover from the loss of volatile, as well as nonvolatile, storage is also stored on stable storage. Failure of permanent storage could be due to disasters either natural or man-made. A manually assisted database regeneration is the only possible remedy to permanent storage failure. However, if multiple generations of archival database are kept, loss of the most recent generation, along with the loss of the nonvolatile storage, can be recovered from, by reverting to the most recent previous generation and, if possible, manually regenerating the more recent data.

Implementation of Stable Storage

Stable storage is implemented by replicating the data on a number of separate nonvolatile storage devices and using a careful writing scheme(described below). Errors and failures occurring during transfer of information and leading to inconsistencies in the copies of data on stable storage can be arbitrated. A mix of HDD and SSD with multiple backups for vital data is the safe practice!

A write to the stable storage consists of writing, two or more times, the same block of data from volatile storage to distinct nonvolatile storage devices. If the writing of the block is done successfully, then all copies of data will be identical and there are no problems. If one or more errors are introduced in one or more copies, then the correct data is assumed to be the copy that has no errors. If two or more sets of copies are found to be error free, but the contents do not agree, than the correct data is assumed to be the set which has the largest number of error free copies. If there are the same number of copies in two or more such identical sets, then, one of these sets is arbitrarily assumed to contain the correct data.

11.1.2 Types of Errors in Database Systems and Possible Detection Schemes

Errors in the use of the database can be traced to one of the following causes: user error, consistency error, system error, hardware error, or external environmental conditions.

User Error: This includes errors in application programs as well as errors made by on-line users of the database. One remedy is to allow on-line users limited access rights to the database, let us say, read, only. Furthermore, any insertion or update operations require that appropriate validation check routines are built into the application programs and that these routines perform appropriate checks on the data entered: the routines will flag any values that are not valid and prompt the user to correct these errors.

Consistency Error: The database system should include routines that check for consistency of data entered in the database. Due to oversight on the part of the DBA, some of the required consistency specifications may be left out which could lead to inconsistency in the stored data. A simple distinction between validity and consistency errors is in order at this time. **Validity** establishes that the data is of the

correct type and within the specified range; consistency establishes that it is reasonable with respect to itself or of the current values other data-items in the database.

System Error: This encompasses errors in the database system or the operating system including situations such as **deadlock**(see section 10.8). Such errors are fairly hard to detect and requires reprogramming the erroneous components of the system software if possible or working with the DBMS vendor. Situations such as deadlocks are catered for in the DBMS by allowing appropriate locking facilities. Deadlocks are also catered to in the operating system by deadlock avoidance, prevention, or detection schemes.

Hardware Failure: This includes hardware malfunctions including storage system failures.

External Environmental Failure: Power failure is one possible type. Others are, for example, fire, flood and other natural disasters, or malicious acts.

In addition to validity checks built into the application programs using a database, the database system usually contains a number of routines to recover from some of the above errors. These routines enforce consistency of the data entered in the database. The required consistencies that are to be enforced are indicated by the DBA.

11.1.3 Audit Trails

The concept of audit trail is not new: recall the Greek myth about Theseus who marked his trail into the labyrinth, where the monster Minotaur lived, using a ball of string given to him by Ariadne. After killing Minotaur, Theseus used the trail marked by the string to find his way out of the labyrinth. Incidentally, the ball of string was magical and it did not run-out on Theseus. The need for the reliability and relative permanency of such a trail is illustrated in the children's story of Hansel and Gretel: a trail marked by bread crumbs was eaten by birds and the pair were unable to find their way back home!

In accounting practice, each transaction is recorded in chronological order in a log which is called a **journal** and the recording process is called **journaling**. Before the **transactions** are actually entered to the appropriate accounts (which in accounting practice is called posting), the transactions are recorded in the journal. The actual recording of the transaction is done in the form of double entry: for each transaction, there are debits (to one or more accounts which are charged) and credits (to one or more accounts which are credited by a positive amount) and the sum of these debits and credits are equal. This double entry helps in detecting errors and ensures the reliability of the accounting records.

The DBMS also has routines which maintain an **audit trail** or a **journal**. An audit trail or a journal is a record of an update operation made on the database. The audit trail records **who** (user or the application program and a transaction number), **when** (time and date), (from) **where** (location of the user and or the terminal) and **what** (identification of the data affected, as well as, a before and an after image of that portion of the database that was affected by the update operation). In addition, a DBMS contains routines which make a backup copy of the data that is modified. This is done by taking a snapshot of the before and after image of that portion of the database that is modified. For obvious reasons, the backups are produced on a separate storage medium**.**

11.1.4 Recovery Schemes

Recovery schemes can be classified as forward or backward recovery. Database systems use the latter schemes to recover from errors.

Forward Error Recovery: In this scheme, when a particular error in the system is detected, the recovery system makes an accurate assessment of the state of the system and then makes appropriate adjustments, based on the anticipated result had the system been error free. The adjustments are obviously dependent on the error and consequently the error types have to be anticipated by the designers of the recovery system. The aim of the adjustment is to restore the system so that the effects of the error are cancelled and the system can continue to operate, as if there had been no errors. This scheme is not applicable to unanticipated errors.

Backward Error Recovery: In this scheme no attempt is made to extrapolate what the state of the system would have been had the error not occurred. Instead, the system is reset to some previous correct state that is known to be free of any errors. The backward error recovery is, as such, a simulated reversal of time and it does not try to anticipate the possible future state of a system.

11.2 Transactions

A single DBMS operation as viewed by an user, for example, update the grade of a student in the relation ENROL (*Student_Name*, *Course*, *Grade*), involves more than one task. Since the data resides on a secondary nonvolatile storage medium, the data will have to be brought into the volatile primary memory for manipulation. This, in turn, requires that the data be transferred between secondary storage and primary storage, the transfer usually performed in blocks of the implementation-specified size. The transfer task consists of locating the block in the secondary storage device containing the required tuple, (which in turn may be preceded by searching an index), obtaining the necessary locks on the block or the tuple involved in the update, and reading-in this block. This task is followed by making the update to the tuple in memory, which in turn is followed by another transfer task, i.e., writing the tuple back to secondary device, and releasing the locks.

In order to reduce the number of accesses to disk, the blocks are read into blocks of main memory, which are called **buffers**. We can, thus, assume that a program performs input/output using, e.g., the *get* and *put* operations, and the system transfers the required block from secondary memory to main memory using the **Read** and **Write** operations. The block read(write) tasks need not be performed in case the system uses buffered input(output) and the required data(space) is already in the primary memory buffer. In such a case the *get*(*put*) operation of the program can input(output) the required data from(to) the appropriate buffer. If the required data is not in the buffer, the buffer manager does a read operation and obtains the required data, after which the data is inputted from the buffer to the program executing the *get* statement. Similarly, if there is no more space left in the buffer, then the *put* operation causes the buffer to be written to the secondary storage (with a **Write**) and, then the *put* operation transfers the data from main memory to the space made available in the buffer.

The above DBMS operation of changing the grade of a student in a given course initiated by a user and appearing to her/him as a single operation, actually requires a number of distinct tasks or steps to be performed by the DBMS and results in a change of a single data item in the database: this is illustrated by a skeleton program given in Figure 11.1.

In this program the comment indicates the definition of the action u**pdate** ENROL of the record for a given student in a given course: this action is being referenced later with the keywords *commit* and *rollback*. The statements defined for the update operation are assumed to modify a temporary copy of the selected portion of the database (the main memory copy of the block of nonvolatile storage containing the tuple for the relation ENROL). Here we are using *error* to indicate whether there are any errors whatsoever during the execution of the statements defined for the action Update ENROL. If there were any errors, we would like to undo any changes made to the database by the statements defined for the Update action. This would involve simply discarding the temporary copy of the affected portion of the database. The database itself is not changed if a temporary copy of the database is being used. In case there were no errors, we would want the changes made by the Update operations to become permanent by being reflected in the actual database.

```
Procedure Modify_Enrol (Student_Name, Course, New_Grade);
   define action update ENROL(Student_Name, Course, Grade)as
            {*action update ENROL is defined as the next two statements*}
    begin
            get for update ENROL where
            ENROL.Student_Name = Student_Name and
            ENROL.Course = Course ;
            ENROL.Grade := New_Grade;
    end
   if error
    then
            rollback action update ENROL;{* do not output ENROL *}
    else
            commit action update ENROL;{* output ENROL *}
   end Modify_Enrol;
```

Figure 11.1. *Modifying a Tuple in the Database*

Figure 11.2 shows the successive states of the database system at different points of the execution of the program of Figure 11.1, with the change of the student Jones grade in course 353, from **in prog**ress to **A**, as shown in Figure 11.2(d). In case there are any errors by the program of Figure 11.1, the program ignores any modifications and the record for Jones remains unchanged as shown in Figure 11.2(e).

The program unit Modify_Enrol given above consists of a number of statements, each of which is executed one at a time (in reality each of the statements of Figure 11.1 in turn is compiled into a number of machine instructions, each of which is executed, one at a time in a sequential manner). Such sequential execution can be interrupted due to errors. (Interrupts to execute the statements of other concurrent programs can also occur, but we will ignore this type of interruption for the time being.) In case of errors, the program may be only partially executed. However, for preserving the consistency of the database we want to ensure that the program is executed as a single unit, the execution of which will not change the consistency of the database. Thus an interruption of a transaction following a system detected error will return the database to its state before the start of the transaction. Such a program unit which operates on

the database to perform a read operation or an update operation (which includes modification, insertion and deletion) is called a ***transaction***.

Definition: A ***transaction*** is a program unit whose execution may change the contents of a database. If the database was in a consistent state before a transaction, then on the completion of the execution of the program unit corresponding to the transaction, the database will be in a consistent state. This in turns requires that the transaction can be considered to be atomic: it is executed successfully or in case of errors, the user can view the transaction as not having been executed at all.

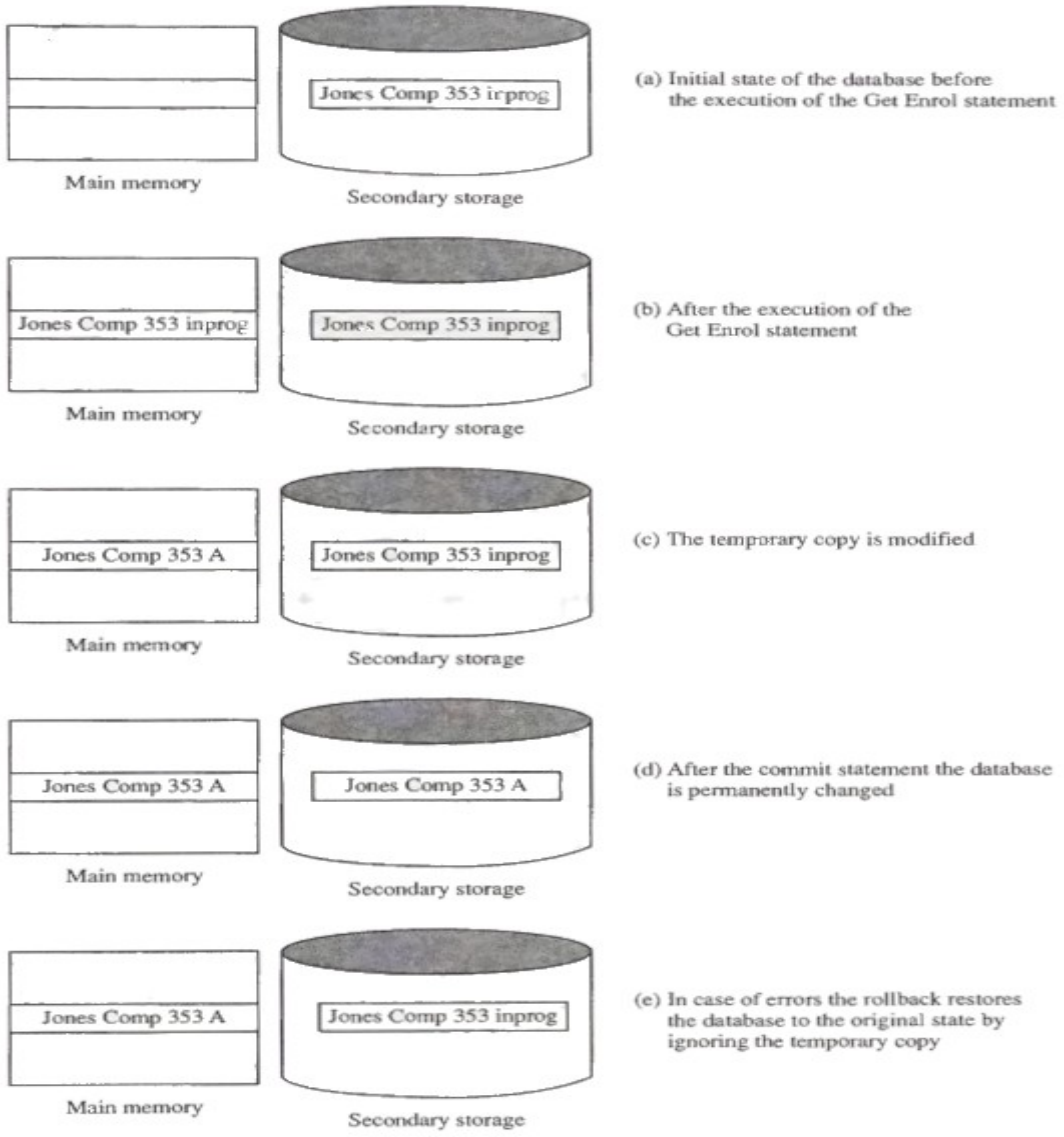

Figure 11.2 *Database states for program of* ***Figure 11.1***

The relationship between an application program and a transaction is shown in Figure 11.3. The application program can be made up of a number of transactions, T_1, T_2, .., T_n. Each such transaction T_i starts at the time T_{istart}. It commits (or rollbacks) at time $T_{icommit}$ ($T_{irollback}$) and terminates at time T_{iend}.

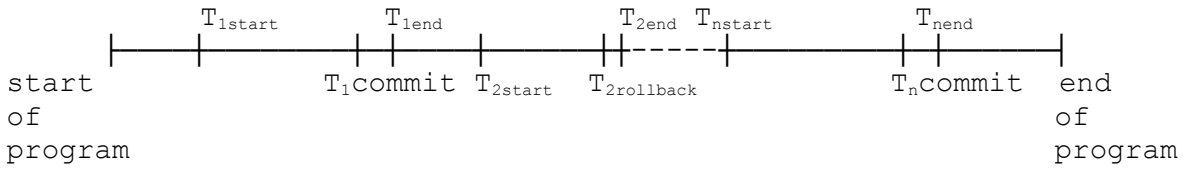

Figure 11.3 *Application program and transactions*

The Commit and Rollback operations included at the end of a transaction are used to ensure that the user can view a transaction as an atomic operation, which preserves database consistency. The commit operation which is executed at the completion of the modifying phase of the transaction allows the modifications made on the temporary copy of the database items to be reflected in the permanent copy of the database (we will defer to a later part of this chapter, the presentation of recovery related operations prior to making changes in the permanent copy of the database). The rollback operation (which is also called the undo operation) is executed if there was an error of some type during the modification phase of the transaction, and indicates that any modifications made by the transaction are ignored; consequently, none of these modifications are allowed to change the contents of the database. If the transaction T_i is rolled back, then the logic of the application program is responsible for deciding whether or not to execute the transaction T_j (for i < j ¾ n). *Once committed, a transaction cannot be rolled back.*

```
Procedure Multiple_Modify Student_Name(Current_name, New_Name);
define action update STUDENT_INFO(Current_Name, New_Name)  as
 begin
   get STUDENT_INFO where Student_Name = Current_name;
   STUDENT_INFO.Student_Name := New_Name;
   end;
define action update ENROL(Current_Name, New_Name)  as
begin
   while no_more_tuples_in  ENROL do;
            get ENROL where ENROL.Student_Name = Current_Name;
            ENROL.Student_Name := New_Name;
            end;
   end;
if error
 then
    rollback action update STUDENT_INFO,   action  update ENROL;
 else
    commit action update STUDENT_INFO,    action  update ENROL;
end Multiple_Modify;
```

Figure 11.4 *Transaction involving multiple modifications*

From the definition of a transaction, we see that the status of a transaction and the observation of its actions must not be visible from outside the transaction until the transaction terminates. Any notification of what a transaction is doing must not be communicated, for instance via a message on to a terminal, until the transaction commits. Once a transaction terminates, the user may be notified of its success or failure.

There could be other DBMS operations which may be viewed by the user as a single action but could involve multiple changes. Consider the operation of changing the name of a student, let us say, from Jones to Smith-Jones. For consistency, the DBMS application program which interfaces with the user must change the name in the relations STUDENT_INFO(*Student_Name*, *Phone_No*, *Major*), corresponding to the student Jones, and all tuples pertaining to this student in the relation ENROL(*Student_Name*, *Course*, *Grade*). A skeleton program to support this is given below in Figure 11.4:

We see from the above skeleton program that modifying the student name involves a number of database accesses and changes. As these changes can only occur one at a time, there is a period of time between the start of the execution of this program and its termination, during which the database is in an inconsistent state. For example, after the appropriate tuple in STUDENT_INFO is changed, we do not have referential integrity, there being no tuple in STUDENT_INFO corresponding to the tuples in ENROL for the student Jones (whose name has just been modified in STUDENT_INFO). Similarly, between the start of the update for the relation ENROL and its completion, some tuples have Smith-Jones as the value for the *Student_Name* attribute and others have Jones.

The point we are trying to illustrate is that a database operation as viewed by a user as a single operation, in fact involves a number of database tasks, and there is no guarantee that the database is in a consistent state between these tasks. However, the user can view these tasks as a single operation (or the so called **atomic** operation), which will complete successfully or not at all. In the former case the changes are made and in the latter case the database remains unchanged. In either case, after the completion of the transaction, the database is in a consistent state.

11.2.1 States of a Transaction

A transaction can be considered to be an atomic operation by the user, however in reality it goes through a number of states during its lifetime. Figure 11.5 gives these states of the transaction, as well as the cause of a transition between these states.

A transaction can end in three possible ways: it can either end after a commit operation (a **successful termination**); or it can detect an error during its processing and decide to abort itself by performing a rollback operation (a **suicidal termination** of the transaction); or the DBMS or the operating system can force it to be aborted for one reason or another (**murderous termination** of the transaction).

We assume that the database is in a consistent state before a transaction starts. A transaction starts when the first statement of the transaction is executed: it becomes active and we assume that it is in the **modify** state. The transaction modifies the database during its modification state. At the end of the modify state, there is a transition of the transaction into one of the following states: **start-to-commit**, **abort**, or **error**. In case the transaction completes the modification state satisfactorily, it enters the start-to-commit state where it instructs the DBMS to reflect the changes made by it into the database. Once **all** the changes made by the transaction are propagated to the database, the transaction is said to be in the

commit state and from there the transaction is **terminated**, the database once again being in a consistent state. In the interval of time between the start-to-commit state and the commit state, some of the data changed by the transaction in these buffers may or may not have been actually propagated to the database on the nonvolatile storage.

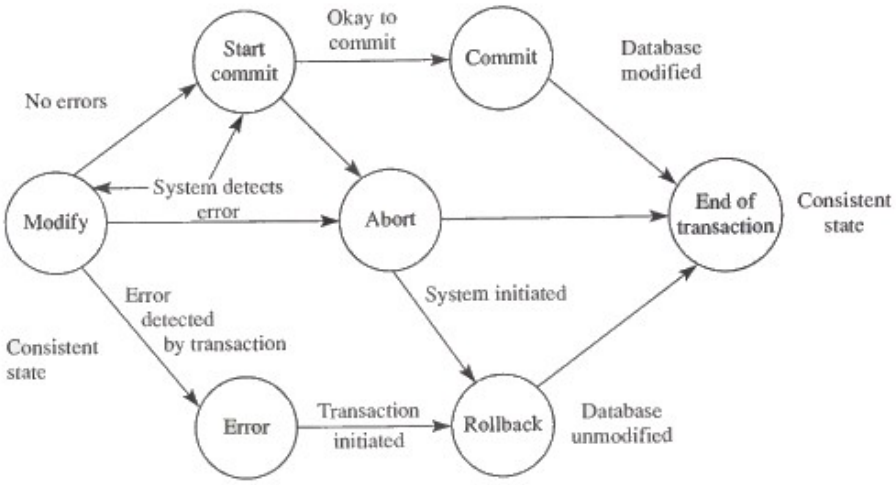

Figure 11.5 *Transaction States*

There is a possibility that all the modifications made by the transaction cannot be propagated to the database due to conflicts or hardware failures. In this case the system forces the transaction to the abort state. The abort state could also be entered from the modify state if there are system errors, for example, division by zero or an unrecoverable parity error. In case the transaction, while in the modify state, detects an error, it decides to terminate itself (suicide) and enters the error state, and, thence, the rollback state**.** If the system aborts a transaction, it may have to initiate a rollback to undo partial changes made by the transaction. An aborted transaction, which had made no changes to the database, is terminated without the need for a rollback, hence there are two paths in Figure 11.5 from the abort state to the end of transaction. A transaction, which on the execution of its last statement, enters the start to commit and from there, the commit state, is guaranteed that the modifications made by it are propagated to the database.

The transaction outcome can be either successful (if the transaction goes through the commit state), suicidal (if the transaction goes through the rollback state) or murdered (if the transaction goes through the abort state) as shown in Figure 11.5. In the last two cases, there is no trace of the transaction left in the database, and only the log indicates that the transaction was ever run.

The transaction outcome can be either successful (if the transaction goes through the commit state), suicidal (if the transaction goes through the rollback state) or murdered (if the transaction goes through the abort state) as shown in Figure 11.5. In the last two cases, there is no trace of the transaction left in the database, and only the log indicates that the transaction was ever run.

Any messages, given to the user by the transaction, must be delayed till the end of a transaction, at which point the user can be notified as to the success or failure of the transaction, and in the latter case, the reasons for the failure.

11.2.2 Properties of a Transaction

From the definition of a transaction, we see that the status of a transaction and the observation of its actions is not visible from outside the transaction until the transaction terminates. Any notification of what a transaction is doing must not be communicated, for instance via a message on to a terminal, until the transaction is terminated. Nor should any partial changes made by an active transaction be visible from outside the transaction. Once a transaction ends, the user may be notified of its success or failure, and the changes made by the transaction are accessible. In order for a transaction to achieve these characteristics, it should have the properties of **atomicity**, **consistency**, **isolation** and **durability**. These properties referred to as the ***ACID*** test(for **a**tomicity, **c**onsistency, **i**solation and **d**urability), represent the transaction paradigm. We amplify the significance of each of these properties in the following paragraphs.

The **atomicity property** of a transaction implies that it will run to completion as an indivisible unit and at the end of which either no changes would have occurred to the database or the database would have been changed in a consistent manner. At the end of a transaction the updates made by the transaction will be accessible to other transactions and the processes outside the transaction.

The **consistency property** of a transaction implies that if the database was in a consistent state before the start of a transaction, then on termination of a transaction the database will also be in a consistence state.

The **isolation property** of a transaction indicates that actions performed by a transaction will be isolated or hidden from outside the transaction until the transaction terminates. This property gives the transaction a measure of relative independence.

The **durability property** of a transaction ensures that the commit action of a transaction, on its termination, will be reflected in the database. The permanence of the commit action of a transaction requires that any failures after the commit operation of a transaction will not cause loss of the updates made by the transaction.

11.2.3 Failure Anticipation and Recovery

In designing a reliable system one tries to anticipate as many different types of failures as one can and provides for the means to recover from these without loss of information. Although, some failures which may be very rare may not be catered to for economic reasons. Recovery from failures which are not thought of, overlooked or ignored may not be possible. In common practice,the recovery system of a DBMS is designed to anticipate and recover from the following types of failure:

Failures without loss of data: This type of failure is due to errors that the transaction discovers before it reaches the start to commit state. It can also be due to the action of the system which resets its state to that which existed before the start of the transaction. No loss of data is involved in this type of failure especially in the case where the transactions are run in a batch mode; these transactions can be rerun at a later point in time in the same sequence.

Failure with loss of volatile storage: Such a failure can occur as a result of software or hardware errors. The processing of an active transaction is terminated in an unpredictable manner before it reaches its commit or rollback state and the contents of the volatile memory are lost.

Failure with loss of nonvolatile storage: This is the sort of failure which can occur due to the failure of a nonvolatile storage system; for example, a head crash on a disk drive, or loss due to errors in writing to a nonvolatile device.

Failure with a loss of stable storage: This fourth type involves loss of data stored on stable storage: the cause of the loss could be due to natural or man-made disasters. Recovery from this type of failure requires manual regeneration of the database. The probability of such a failure is reduced to a very small value by having multiple copies of data in the stable storage, stored in physically secure environments in geographically dispersed locations.

11.3 Recovery in a Centralized DBMS

The basic technique to implement the database transaction paradigm, in the presence of failures of various kinds, is by using data redundancy in the form of **logs**, **checkpoints** and **archival** copies of the database.

11.3.1 Logs

The log which is usually written onto stable storage, contains the redundant data required to recover from volatile storage failures and also from errors discovered by the transaction or the database system. For each transaction the following data is recorded on the log:

- The start-of-transaction marker.
- The transaction identifier which could include the who and where information referred to above in Section 11.1.3.
- The record identifiers which include the identifiers for the record occurrences (tuple identifier in the case of relations).
- The operation(s) performed on the tuples (insert, delete, modify).
- The previous value(s) of the modified data. This information will be required for undoing the changes made by a partially completed transaction, and is called the UNDO log. In the case where the modification made by the transaction is the insertion of a new record, the previous values can be assumed to be null.
- The updated value(s) of the modified tuple(s). This information will be required for making sure that the changes made by a committed transaction are in fact reflected in the database and can be used to redo these modifications. This information is called the REDO part of the log. In case the modification made by the transaction is the deletion of a record, the updated values can be assumed to be null.
- A commit transaction marker if the transaction is committed; otherwise an abort or rollback transaction marker.

The log is written before any updates are made to the database. This is called the **write-ahead-log strategy**. In this strategy a transaction is not allowed to modify the physical database until the undo portion of the log (i.e. the portion of the log which contains the previous value(s) of the modified data) is written to stable storage. Furthermore, the log write-ahead strategy requires that a transaction is allowed to commit only after the redo portion of the log, along with the commit transaction marker is written onto the log. In effect, both the undo and redo portion of the log will be written onto stable storage before a transaction commit. Using this strategy, the partial updates made by an uncommitted transaction can be

undone using the undo portion of the log, and a failure occurring between the writing of the log and the completion of updating the database corresponding to the actions implied by the log can be redone.

Let us see how the log information can be used in case of a system crash, with the loss of volatile information. Consider a number of transactions, as shown in Figure 11.6. The figure shows the system start off at time t_0 and a number of concurrent transactions T_0, T_1, ..., T_{i+6} are made on the database. Suppose a system crash occurs at time t_x.

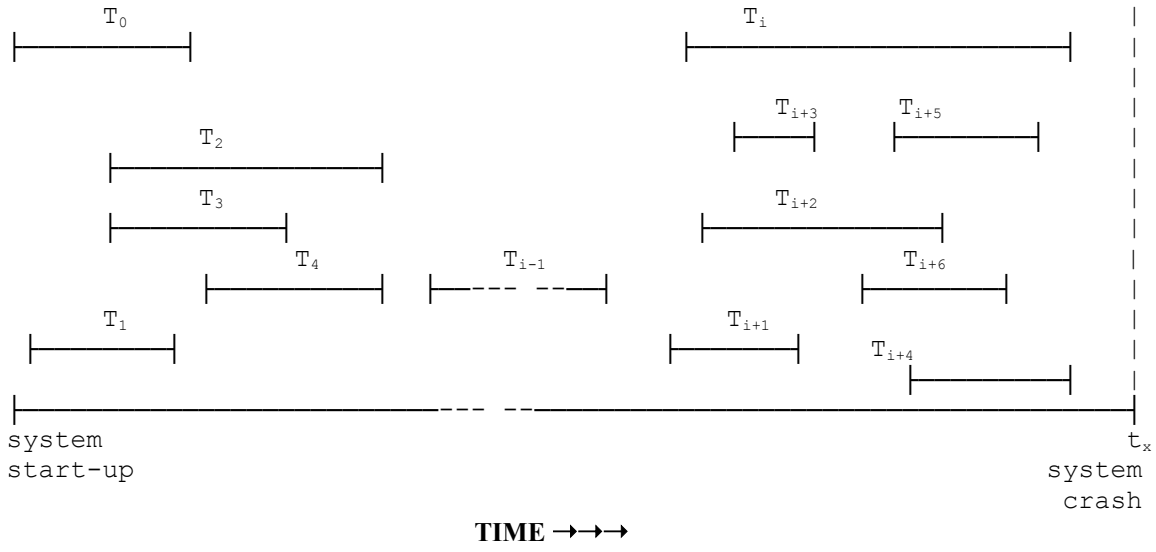

Figure 11.6 *DBMS Operation to a System Crash*

We have stored the log information for transactions T_0 through T_{i+2} on stable storage, and we assume that this will be available at the time the system comes up after the crash. Furthermore, we assume that the database existing on the nonvolatile storage will also be available. It is clear that the transactions which were not committed at the time of the system crash will have to be undone. The changes made by these uncommitted transactions will have to be rolled back. The transactions which have not been committed can be found by examining the log, and those transactions which have a Start of transaction marker but no commit or abort transaction marker are considered to have been active at the time of the crash. These transactions have to be rolled back to restore the database to a consistent state. In Figure 11.6 the transactions T_i, and T_{i+6} had started before the crash, but they had not been committed, and, hence, are undone.

However, it is not clear from the log as to what extent the changes made by committed transactions have been actually propagated to the database on the nonvolatile storage. The reason for this uncertainty is the fact that buffers (implemented in volatile storage) are used by the system to hold the modified data. Some of the changed data in these buffer may or may not have been actually propagated to the database on the nonvolatile storage. In the absence of any method of finding out the extent of the loss, we will be forced to redo the effects of all committed transactions. For the example of Figure 11.6, this involves redoing the changes made by all transactions from time t_0. Under such a scenario, the longer the system operates without a crash, the longer it will take to recover from the crash.

In the above, we have assumed that the log information is available up to the time of the system crash in nonvolatile storage. However, the log information is also collected in buffers. In case of a system crash with loss of volatile information, the log information, being collected in buffers will also be lost and, hence, transactions which had completed for some period prior to the system crash may be missing their respective end of transaction markers in the log. Such transactions if rolled back, will likely be only partially undone. (Why ?). The write-ahead-log strategy avoids this type of recovery problem, since the log information is forced to be copied to stable storage before the transaction commits.

These problems point to the conclusion that some means must be devised such that all the log information, as well as modifications to the database existing at a given point in time, is propagated to stable storage at regular intervals so that the recovery operation after a system crash does not have to re-process all transactions from the time of a start-up of the system.

11.3.2 Checkpoints

In an on-line database system, for example an airline reservation system, there could be hundreds of transactions being handled per minute. The log information for this type of database will contain a very large volume of information. A scheme called checkpoint is used to limit the volume of log information that has to be handled and processed in the event of a system failure involving the loss of volatile information. The checkpoint scheme is an additional component of the logging scheme described above.

In the case of a system crash with loss of volatile information, the log information being collected in buffers will be lost. A checkpoint operation, performed periodically, copies this type of information onto stable storage. The information and operations performed at each checkpoint consist of the following:

- A start-of-checkpoint record giving the identification that it is a checkpoint along with the time and date of the checkpoint. This checkpoint record is written to the log on stable storage device.
- Copy to the log on stable storage all log information from the buffers in the volatile storage.
- Propagate all database updates from the buffers in the volatile storage to the physical database.
- An end-of-checkpoint record is written and the address of the checkpoint record is saved on a file which will be accessible to the recovery routine on startup after a system crash.

For all transaction, active at checkpoint, their identifiers and their database modification actions, which at that time are reflected only in the database buffers, will be propagated to the appropriate storage.

The frequency of checkpointing is a design consideration of the recovery system. A checkpoint can be taken at fixed intervals of time (let us say every 15 minutes). If this approach is used, a choice has to be made, regarding what to do with the transactions that are active when the checkpoint signal is generated by a system timer. In one alternative called ***transaction consistent checkpointing***, the transactions that are active when the system timer signals a checkpoint, are allowed to complete, but no new transactions (requiring modifications to the database) are allowed to be started until the checkpoint is completed. This scheme, though attractive, makes the database unavailable at regular intervals and may not be acceptable for certain on-line applications. In addition, this approach is not appropriate for long transactions. In the second variation called ***action consistent checkpointing***, active transactions are allowed to complete the current step before the checkpoint and no new actions can be started on the database until the checkpoint is completed; during the checkpoint no actions are permitted on the database. Another alternative called ***transaction oriented checkpointing*** is to take a checkpoint at the end of each transaction by effectively forcing the log of the transaction onto stable storage. In effect, each commit transaction is a checkpoint.

How does the checkpoint information help in recovery? To answer this question, let us reconsider the set of transactions of Figure 11.6, shown below in Figure 11.7, with the addition of a checkpoint being taken at time t_c.

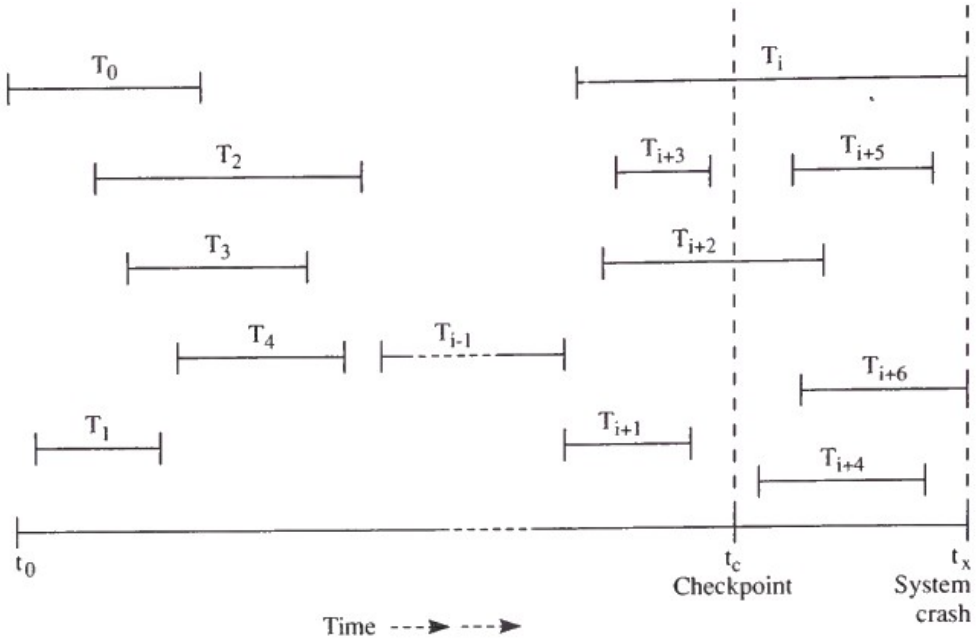

Figure 11.7 *Checkpointing*

Suppose, as before, the crash occurs at time t_x. Now the fact that a checkpoint was taken at time t_c indicates that at that time all log and data buffers were propagated to storage. Transactions T_0, .., T_{i-1}, as well as the transactions T_{i+1} and T_{i+3} were committed, and their modifications are reflected in the database; these transactions are not required to be redone during the recovery operation following a system crash occurring after time t_c with the checkpoint scheme. A transaction, such as T_i (which started before checkpoint time t_c), as well as the transaction T_{i+6} (which started after the checkpoint time t_c), were not committed at the time of the crash, and have to be rolled back. Transactions such as T_{i+4} and T_{i+5} which started after the checkpoint time tc and committed before the system crash have to be redone. Similarly, transactions such as T_{i+2}, which started before the checkpoint time and committed before the system crash, will have to be redone.

Let us now see how the system can perform a recovery at time t_x. Suppose all transactions that had started before the checkpoint time, but not committed at that time, as well as the transactions started after the checkpoint time are placed in an **Undo** list, which is a list of transactions which have to be undone. The Undo list for the transactions of Figure 11.7 is given below:

UNDO List: (T_i, T_{i+2}, T_{i+4}, T_{i+5}, T_{i+6})

Now the recovery system scans the log in a backward direction from the time t_x of system crash. If it finds that a transaction in the Undo list has committed, then that transaction is removed from the Undo list and placed in another list called **Redo** list. The redo list contains all the transactions that have to be redone. The reduced Undo list and the Redo list for the transactions of Figure 11.7 are given below:

REDO List: (T_{i+4}, T_{i+5}, T_{i+2})

UNDO List: (T_i, T_{i+6})

Obviously, all transactions that committed before the checkpoint time need not be considered for the recovery operation. In this way the amount of work required to be done for recovery from a system crash is reduced. Without the checkpoint scheme, the Redo list will contain all transactions except Ti and Ti+6. A system crash occurring during the checkpoint operation, requires recovery to be done using the most recent previous checkpoint.

The recovery scheme described above takes a pessimistic view about what has been propagated to the database at the time of a system crash with loss of volatile information. Such pessimism is adopted both for transactions committed after a checkpoint, as well as for transactions not committed since a checkpoint. It assumes that the transactions committed since the checkpoint have not been able to propagate their modifications to the database and the transactions still in progress have done so!

Note that in some systems, the term checkpoint is used to denote a correct state of system files, recorded explicitly in a backup-file and, thence, the term checkpointing is used to denote a mechanism used to restore the system files to a previous consistent state. However, in a system that uses the transaction paradigm, checkpoint is a strategy to minimize the search of the log, and the amount of undo and redo required to recover from a system failure with loss of volatile storage.

11.3.3 Archival Database and Implementation of the Storage :Hierarchy of a Database System

Figure 11.8 gives the different types of storage used in a database system. These storage types are sometimes called the storage hierarchy. It consists of the following categories of data: archival database, physical database, archival log, and current log. The data contained in each of these categories and their usage is described below:

● **Physical Database**: This is the on-line copy of the database that is stored in nonvolatile storage and is used by all active transactions.

● **Current Database**: The current version of the database is made up of the physical database, plus modifications implied by buffers in the volatile storage.

● **Archival Database in Stable Storage:** This is the copy of the database at a given point in time, stored onto stable storage. It contains the entire database**,** in a **quiescent** mode (i.e. no transactions were active at the time when the database was copied to the stable storage) and could have been made by simple dump routines to dump the physical database (which in quiescent state would be the same as the current or the on-line database) onto stable storage. The purpose of the archival database is to recover from failures that involve loss of nonvolatile storage. The archiving process is a relatively time-consuming operation and during this period, the database is not accessible. Consequently, archiving is done at very infrequent intervals. The frequency of archiving is then a trade-off between the cost of archiving and that of recovery with the probability of a loss of nonvolatile data being the arbitrator. All transactions that have been executed on the database from the time of archiving have to be redone in a global recovery operation. No undoing is required in the global recovery operation since the archival

database is a copy of the database in a quiescent state, and only the committed transactions since the time of archiving are applied to this database.

● **Current Log**: This contains the log information (including the checkpoint) required for recovery from system failures involving loss of volatile information.

● **Archival Log**: This log is used for failures involving loss of nonvolatile information. The log contains information on all transactions made on the database from the time of the archival copy. This log is written in a chronological order. The recovery from loss of nonvolatile storage uses the archival copy of the database and the archival log to reconstruct the physical database to the time of the nonvolatile storage failure.

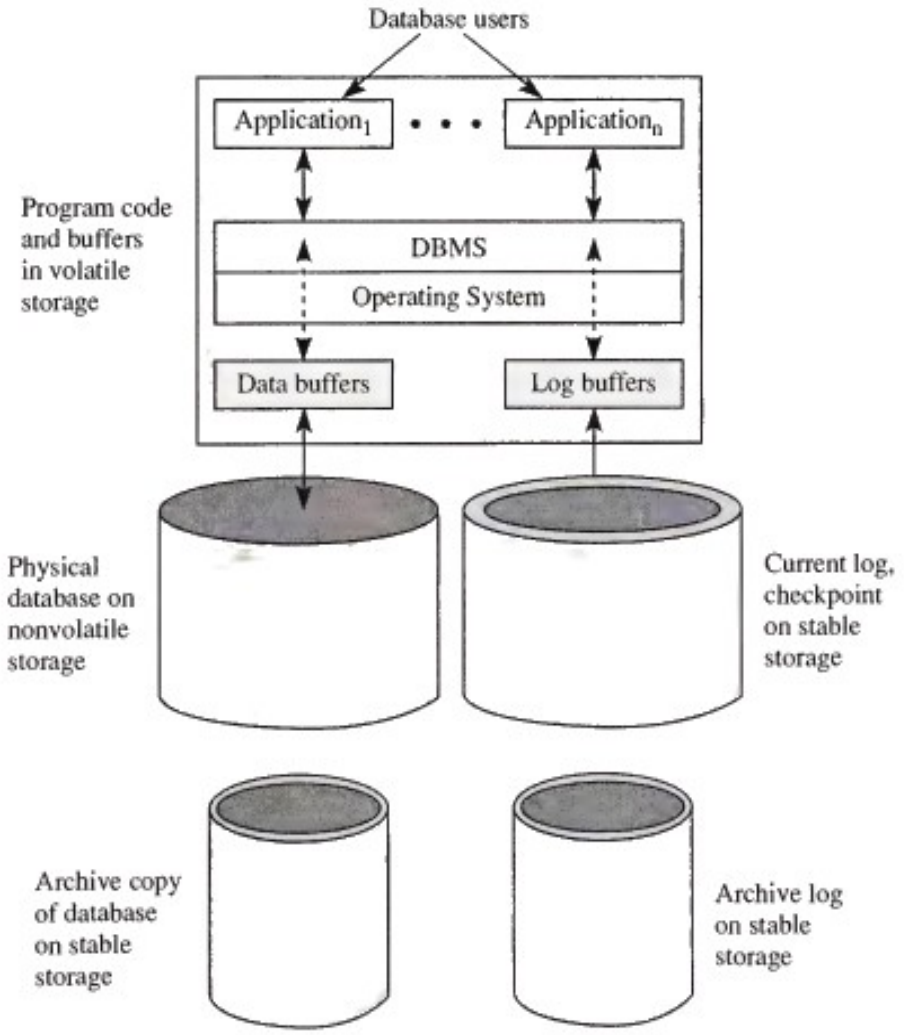

Figure 11.8 *Database storage hierarchy*

With the above storage hierarchy of a database, we can use the following terms to denote different combinations of this hierarchy.

The ***On-line*** or ***current database*** is made up of all the records (and the auxiliary structures such as indices) that are accessible to the DBMS during its operation. The current database consists of the data stored in nonvolatile storage (physical database), as well as the data stored in buffers (in the volatile storage) and not yet propagated to the nonvolatile storage.

The ***materialized database*** is that portion of the database that is still intact after a failure. All the data stored in the buffers would have been lost and some portion of the database would be in an inconsistent state. The log information is to be applied to the materialized database by the recovery system to restore the database to as close a state as possible to the on-line database prior to the crash. Obviously, it will not be possible in all cases to return to exactly the same state as the pre-crash on-line database. The intent is to limit the amount of lost data and the loss of completed transactions,

11.3.4 Do, Undo and Redo

A transaction on the current database transforms it from the current state to a new state. This is the so called DO operation. The undo and redo operations are functions of the recovery subsystem of the database system which are used in the recovery process. The undo operation undoes or reverses the actions (possibly partially executed) of a transaction and restores the database to the state that existed before the start of the transaction. The redo operation redoes the action of a transaction and restores the database to the state it would be at the end of the transaction. The undo operation is also called into play when a transaction decides to terminate itself (suicidal termination). Figure 11.9 shows, graphically, the transformation of the database as a result of a transaction do, redo, and undo.

The undo and redo operations for a given transaction are required to be **idempotent**; that is, for any transaction performing one of these operations once, is equivalent to performing it any number of times. Thus:

Undo(any action) Undo(Undo(.. Undo(any action) ..))

Redo(any action) Redo(Redo(.. Redo(any action) ..))

The reason for the requirement that undo and redo be idempotent is that the recovery process, while in the process of undoing or redoing the actions of a transaction, may fail, without a trace, and this type of failure can occur any number of times before the recovery is completed successful.

Transaction Undo

A transaction that discovers an error while it is in progress and consequently needs to abort itself and rollback any changes made by it, uses the **transaction undo** feature. A transaction also has to be undone when the DBMS forces the transaction to abort. A transaction undo removes all database changes, partial or otherwise, made by the transaction.

Transaction Redo

Transaction redo involves performing the changes made by a transaction that had committed before a system crash. With the write-ahead-log strategy, a committed transaction implies that the log for the transaction would have been written to nonvolatile storage, but the physical database may or may not have been modified before the system failure. A transaction redo modifies the physical database to the new values for a committed transaction. Since the redo operation is idempotent, redoing the partial or complete modifications made by a transaction to the physical database will not pose a problem for recovery.

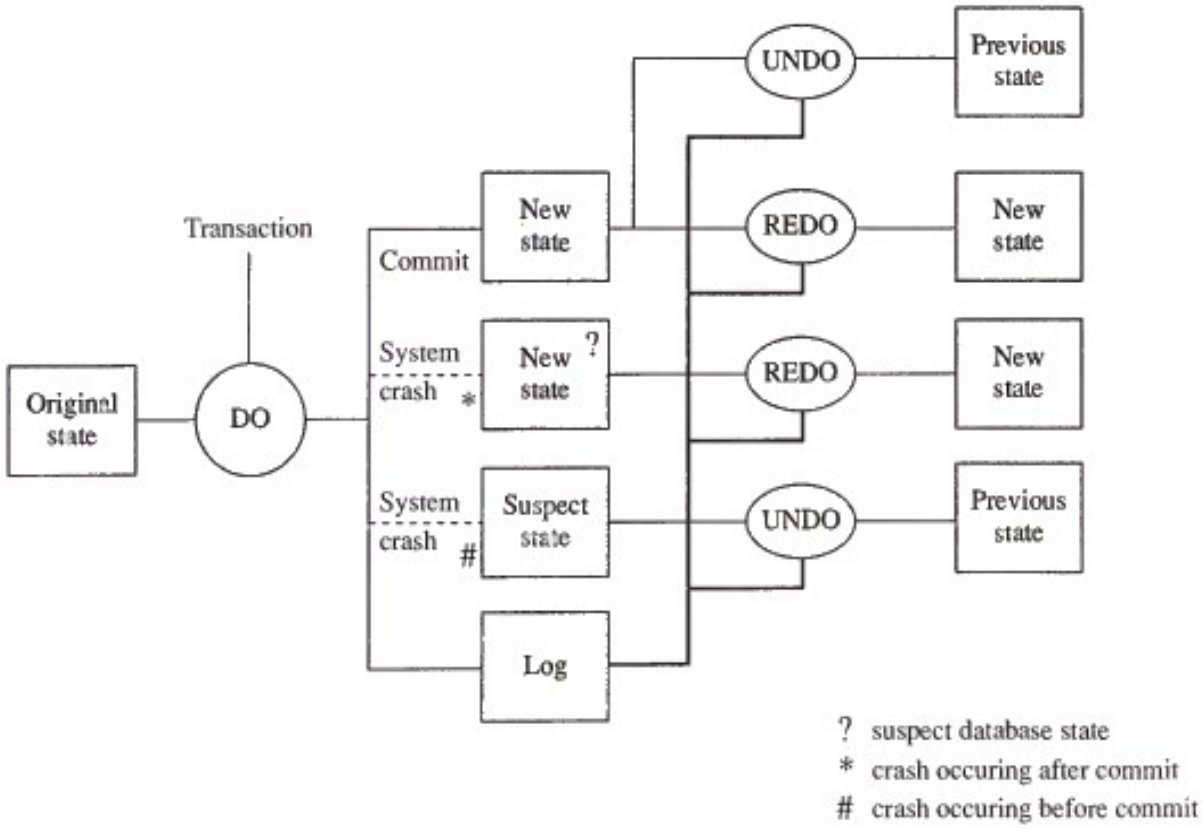

Figure 11.9 *Do, Undo, and Redo operations*

Global Undo

Transactions which are partially complete at the time of a system crash with loss of volatile storage, need to be undone by undoing any changes made by the transaction. The **global undo** operation, initiated by the recovery system, involves undoing the partial or otherwise updates made by all uncommitted transactions at the time of a system failure.

Global Redo

The **global redo** operation is required for recovery from failures involving nonvolatile storage loss. The archival copy of the database is used and all transactions committed since the time of the archival copy are redone to obtain a database updated to a point in time as close as possible to the time of the nonvolatile storage loss. The effects of the transaction in progress at the time of the nonvolatile loss will not be reflected in the recovered database. The archival copy of the database could be anywhere from months to days old and the number of transactions that have to be redone could be quite large. The log for the committed transactions needed for performing a global redo operation have to be stored on stable storage so that these are not lost with the loss of nonvolatile storage containing the physical database.

11.4 Reflecting Updates onto the Database and Recovery

Let us assume that the physical database at the start of a transaction is equivalent to the current database, i.e., all modifications have been reflected in the database on the nonvolatile storage. Under this assumption, whenever a transaction is run against a database, we have a number of options as to the strategy that will be followed in reflecting the modifications made by a transaction as it is executed. The strategies we will explore are the following:

- **Update in place**: in this approach the modifications appear in the database in the original locations and, thus, in case of a simple update, the new values will replace the old values.

- **Indirect update with careful replacement**: In this approach the modifications are not made directly on the physical database. There are two possibilities which can be considered. The first scheme, called **shadow page scheme**, makes the changes on a copy of that portion of the database which is being modified. The other scheme is called update via log and in this strategy of indirect update, the update operations of a transaction are logged and the log of a committed transaction is used to modify the physical database.

In the following sections, we will examine these update schemes in greater detail.

11.4.1 Update in place

In this scheme, (Figure 11.10) the transaction updates the physical database and the modified record replaces the old record in the database on nonvolatile storage. However, the write-ahead-log strategy is used and the log information about the transaction modifications are written before the corresponding *put*(x) operation, initiated by the transaction, is performed. Recall that the write-ahead-log strategy has the following requirements:

(i) before a transaction is allowed to modify the database, at least the undo portion of the transaction log record is written to the stable storage;
(ii) a transaction is committed only after both the undo and the redo portion of the log are written to stable storage.

The sequence of operations for transaction T and the actions performed by the database are shown in Figure 11.11. The initiation of a transaction causes the start of the log of its activities; a Start transaction

along with the identification of the transaction is written out to the log. During the execution of the transaction, any output (in the form of a *put* by the transaction) is preceded by a log output to indicate the modification being made to the database. This output to the log will consist of the record(s) being modified, old values of the data items in the case of an update, and the new values of the data items. The old values will be used by the recovery system to undo the modifications made by a transaction in case a system crash occurs before the completion of the transaction. In case of a system crash occurring after a transaction commits, the new values will be used by the recovery system to redo the changes made by the transaction and thus ensure that the modifications made by a committed transaction are correctly reflected in the database.

Let us consider a transaction shown in Figure 11.11, which consists of reading in the value of some data item X, and modifying it by a certain amount. The transaction then reads in the value of another data item Y and modifies it by an equal but opposite amount. The transaction may subtract, let us say, a quantity **n** from the inventory for part P_x and add this amount to quantities of that item shipped to customer C_y. For consistency this transaction must be completed atomically. A system crash occurring at any time before time t_9 will require that the transaction be undone. A system crash occurring after t_9, when the commit transaction marker is written to the log requires that we redo the transaction to ensure that all of the changes made by this transaction are propagated to the physical data base.

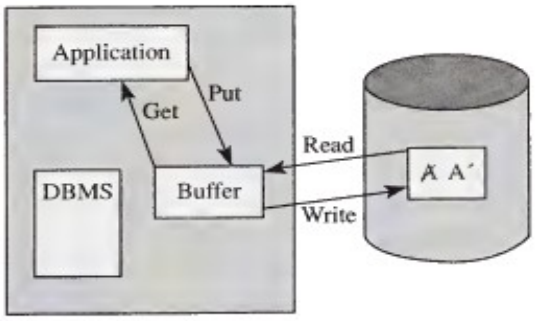

Figure 11.10 *Update in place Scheme*

According to the write-ahead-log strategy, the redo portion of the log need not be written to the log until the commit transaction is issued by the program performing the transaction. However, for simplifying the log, we are combining the undo and redo portions of each modification made by a transaction in one log entry.

Consider another example where a program executes a number of transactions involving a number of distinct records**.** In this case, the transaction atomicity requirement is critical**.** The example involves projects and parts used by the project and inventory of the parts. Suppose we have a number of parts: $Part_1$, $Part_2$, ..., and a number of projects: $Proj_1$, $Proj_2$, ...,. Each project $Proj_i$ uses parts { .., $Part_k$, ..}. Suppose the database contains the following relations:

PART(*Part#*, *Quantity_in_Stock*)
PROJECT(*Project#*, *Part#*, *Quantities_to_Date*)

Time	Transaction Step	Log Operation	Database Operation
t_0	Start of T	**Write**(start Transaction T)	
t_1	*get*(X)		**Read**(X)
t_2	*modify*(X)		
t_3	*put*(X)	**Write**(record X modified, old value of X, new value of X)	
t_4			**Write**(X)
t_5	*get*(Y)		**Read**(Y)
t_6	*modify*(Y)		
t_7	*put*(Y)	**Write**(record Y modified, old value of X, new value of X)	
t_8			**Write**(Y)
t_9	Start Commit	**Write**(Commit transaction T);	
t_{10}	End of T		

Figure 11.11 *Direct Update (Write-ahead-log)*

Consider the execution of the program of Figure 11.12(a) which transfers 100 units of parts $Part_4$ to project $Proj_5$ and 10 units of parts $Part_1$ to project $Proj_2$. Here, each such transfer is considered as a separate transaction, and if the quantity in stock of a part is less than the required quantities to be transferred, then an error condition is said to exist and such a transaction is aborted (a suicidal end). The transfer of x quantity of $Part_i$ from inventory to project $Proj_j$ is considered to be a single atomic operation which either succeeds and performs the appropriate transfer; or, the transaction fails, in which case it does not leave a trace of partial execution (except in the log).

With the update-in-place scheme, the new value of a record field overwrites the old value as shown in Figure 11.12(b). If a transaction involves multiple changes, a system crash occurring before the last modification can be propagated to the database would cause the database to end up in an inconsistent state.

The update-in-place method of updates goes against the well established accounting practice, wherein each and every transaction is recorded, and data is never overwritten. In accounting practice, a compensating transaction is used to make corrections when an error is discovered; and the fact that an error was made is also recorded.

Let us now see how the log information can be used in the recovery process, if a system crash occurs before all the modifications made by a transaction are propagated to the database. Suppose that before the

program was run, the inventory for parts $Part_1$ and $Part_4$ were 400 and 600 respectively; the quantity used by project $Proj_5$ of part $Part_1$ was 100 and the quantity used by project $Proj_2$ of part $Part_4$ was 50.

```
Program: Transfer_parts( input,output);
  var (* declarations are not given but should include all variables as well as database records to be
used                                    and the corresponding local declarations *)

 Procedure many_transactions
   begin
    while not EOF do
     error := false;
     readln(projno, partno, quant);
     start_transaction(modifymode)
     get PART where Part_Number = partno;
     Quantity_in_Stock:= Quantity_in_stock - quant;
     if Quantity_in_Stock < 0
            then error := true
            else begin
                    put PART;
                    get Project where Project_Number  = projno
                    &  Part_number = partno;
                    Quantity_to_Date:= Quantity_to_Date + quant;
                    put PROJECT;
                    end;
     if error
            then  abort_transaction
            else commit_transaction;
     end_transaction;
     end (* while *)
   end (* procedure *)
 end.
```

Figure 11.12(a) *Multiple Direct Updates*

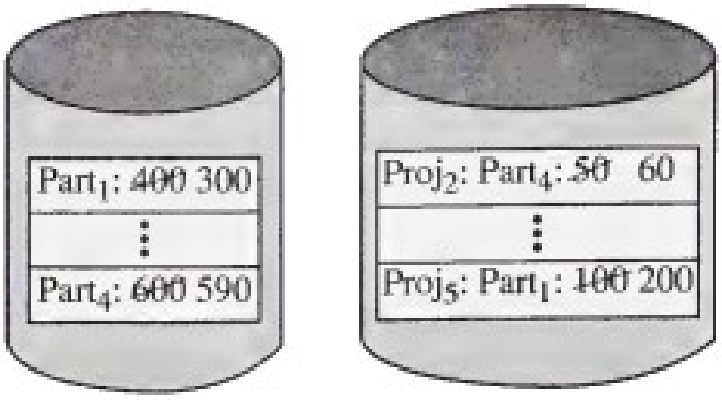

Figure 11.12(b) *Modifications with Update-in-place scheme*

Consider that the program of Figure 11.12(a) was run to transfer 100 units of $Part_1$ from inventory for use in $Proj_5$, followed by the transfer of 10 units of part $Part_4$ from inventory to $Proj_2$. The operations performed by the program are shown in Figure 11.13. The first operation is called transaction T_0; the second operation, T_1. Note: *Quantity_in_Stock* is abbreviated as Q_in_S, and *Quantity_to_Date* as Q_to_D.

Step	Transaction Action	Log Operation	Database Operation
s_0	Start of T_0	**Write**(start Transaction T_0)	
s_1	*get*($Part_1$)		**Read**($Part_1$)
s_2	modify(Q_in_S from 400 to 300)		
s_3	*put* ($Part_1$)	**Write**(record for Part#=$Part_1$, old value of Q_in_S:400, new value of Q_in_S:300)	
s_4			**Write**($Part_1$)
s_5	*get*($Proj_5$)		**Read**($Proj_5$)
s_6	modify(Q_to_D from 100 to 200)		
s_7	*put*($Proj_5$)	**Write**(record for Project#=$Proj_5$, old value of Q_to_D:100, new value of Q_to_D:200)	
s_8			**Write**($Proj_5$)
s_9	Start Commit	**Write**(Commit transaction T_0);	
s_{10}	End of T_0		
s_{10}'	Start of T_1	**Write**(start Transaction T_1)	
s_{11}	*get*($Part_4$)		**Read**($Part_4$)
s_{12}	modify(Q_in_S from 600 to 590)		
s_{13}	*put* ($Part_4$)	**Write**(record Part#=$Part_4$, old value of Q_in_S:600, new value of Q_in_S:590)	
s_{14}			**Write**($Part_4$)
s_{15}	*get*($Proj_2$)		**Read**($Proj_2$)
s_{16}	modify(Q_to_D from 50 to 60)		
s_{17}	*put*($Proj_2$)	**Write**(record Project#=$Proj_2$, old value of Q_to_D:50, new value of Q_to_D:60)	
s_{18}			**Write**($Proj_2$)
s_{19}	Start Commit	**Write**(Commit transaction T_1);	
s_{20}	End of T_1		

Figure 11.13 *The steps for two transactions*

Now suppose that while the program of Figure 11.12(a) was executing, there was a system crash with loss of volatile storage. Let us consider the various possibilities as to the progress made by the program and the sequence of recovery operations required using the information from the write-ahead-log.

If the crash occurs just during or after step s_4, then the log would have the following information for the transaction T_0:

Start of T_0
  record Part#=$Part_1$,
    old value of Q_in_S:400
    new value of Q_in_S:300

The recovery process, when it examines the log, finds that the commit transaction marker for T_0 is missing and, hence, will undo the partially completed transaction T_0. To do this it will use the old value for the modified field of the part record identified by $Part_1$ to restore the *Quantity_in_Stock* field of the part record for $Part_1$ to the value 400, and, hence, the database to the consistent state that existed before the crash and before transaction T_0 was started. If the crash occurs after step s_9 is completed, then the recovery system will find an end-of-transaction marker for transaction T0 in the log, and the log entry would be as given below:

Start of T_0
  record Part#=$Part_1$,
    old value of Q_in_Stock:400
    new value of Q_in_Stock:300
  record Project#=$Proj_5$
    old value of Q_to_D:100,
    new value of Q_to_D:200
  Commit T_0

However, since the log was written ahead of the database, all modifications to the database may not have been propagated to the database. Thus, the recovery system, to ensure that all modifications made by the transaction T_0 are propagated to the database, will redo the committed transaction. To do this it uses the new values of the appropriate fields of the records identified by Part#=$Part_1$ and Project#=$Proj_5$. This will restore the database to an up-to-date state, with the modifications of the committed transactions having been propagated to the database.

It is obvious that if the system crash occurs after step $s_{10'}$, but before step s_{19}, then the recovery operation will require the undoing of modifications made by transaction T_1 and redoing those made by transaction T_0. Similarly, a crash occurring any time after step s_{19} will require the redoing of the modifications made by both transactions T_0 and T_1.

It is important to point out that the key to the recovery operation is the log, which is written on to stable storage ahead of the update-in-place of the database, and, hence, the log information survives any crash. However, the writing of the log may itself be interrupted by a system crash and log information may be incomplete. If the crash occurs sometime during step s_9, the commit transaction marker for transaction T_0, may not be safely written on the log, and this implies that the recovery system will undo the transaction even if all the modifications made by transaction T_0 have been propagated to the database.

In the above example, we have assumed that the DBMS, propagates the modifications to the database as soon as the log entry for the modifications are written to stable storage. However, if the database system defers the propagation to the database until commit step for the transaction, then in the event of a system crash the recovery tasks are modified slightly(see exercise 11.17). If the transaction is rolled back by the user program, the rolling back operation involves writing a rollback marker to log and inhibiting the propagation of the changes to the database. The propagation to the database will also be inhibited if the transaction is aborted by the system before it commits; the last log entry in that case would be an abort transaction marker.

In either of the two possible choices of propagating the changes to the database, the consistency criterion of the database requires that that portion of the database being modified by a transaction is accessible, exclusively to the transaction, for the duration of the transaction.

11.4.2 Indirect Update and Careful Replacement

In the indirect update and careful replacement scheme, the database is not directly modified, but a copy is made of that portion of the database which is to be modified, and all modifications are made on this copy. Once the transaction commits, the modified copy replaces the original.

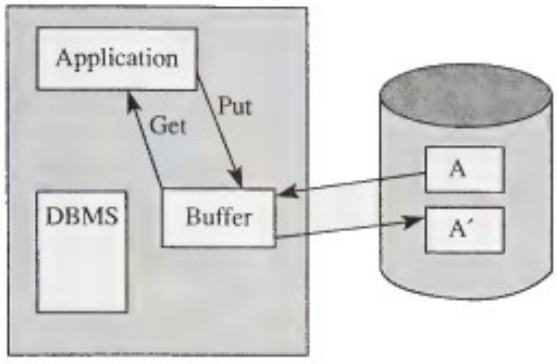

Figure 11.14 *Indirect Page Allocation Scheme*

In the most common scheme used, the indirect page allocation scheme, the modifications to the database are directed to new blocks(page) on nonvolatile storage (Figure 11.14). Each such new block is a copy of the database block containing the records being modified. The old block of the database remains intact. When the transaction commits, the new blocks can be used to replace the old blocks in an atomic manner. In the case of a system crash, the old blocks are still available and the recovery operation is simplified.

In another form of indirect update, no changes are made to the database during a transaction. However, the modified values are written on to a log on stable storage (recall the journal concept of accounting). When the transaction commits, the log is used to write the modifications onto the database. In this case, the rollback of a transaction entails discarding the log entries for the transaction. The recovery operation of a transaction is limited to redoing the modifications made by a transaction which is

recorded in the log entry for that transaction. The undo recovery operation for the transaction has no need of undoing any changes as far as the database on the nonvolatile storage is concerned since no changes were made for an uncommitted transaction.

Reflecting Updates onto the Database and Recovery: Shadow Page Scheme

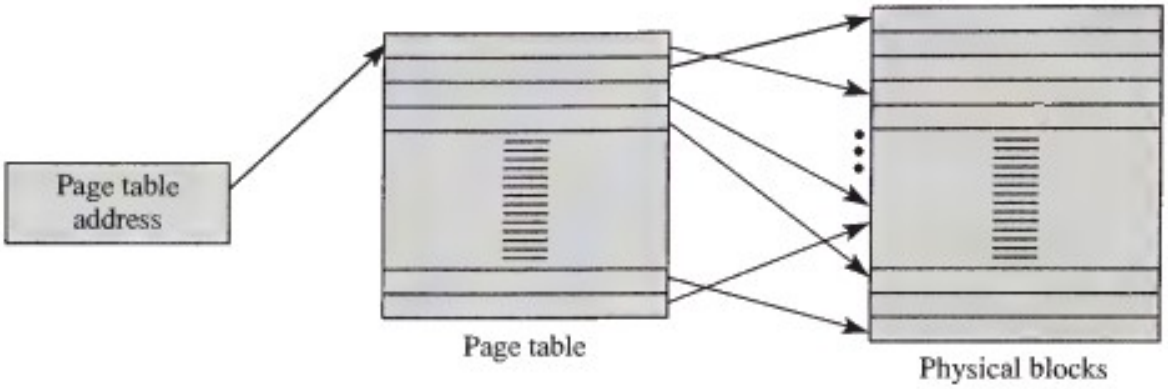

Figure 11.15 *Paging Scheme*

The shadow page scheme is one possible form of the indirect page allocation. Before we discuss this scheme, let us briefly review the paging scheme as used in operating system for virtual memory management. The memory that is addressed by a process (a program in execution is a process) is called virtual memory and it is divided into pages which are assumed to be of a certain size, let us say, 1024(1K) bytes, or more commonly 4096(or 4K) bytes. The logical pages are mapped onto physical memory blocks of the same size as the pages, and the mapping is provided by means of a table known as a page table. The page table, shown in Figure 11.15, contains one entry for each logical page of the process's logical address space. With this scheme, the consecutive logical pages need not be mapped onto consecutive physical blocks.

In the shadow page scheme, the database is considered to be made up of logical units of storage called pages. The pages are mapped into physical blocks of storage (again of the same size as the logical pages) by means of a Page Table, there being one entry for each logical page of the database. This entry contains the block number of the physical storage where this page is stored.

The shadow page scheme, shown in Figure 11.16, uses two page tables for a transaction that is going to modify the database. The original page table is called the *shadow page table*, and the transaction addresses the database using another page table known as the *current page table*. Initially, both the page tables point to the same blocks of physical storage. The current page table entries may change during the life of the transaction. The changes are made whenever the transaction modifies the database by means of a write operation to the database. The pages that are affected by a transaction are copied on to new blocks of physical storage and these blocks, along with the blocks not modified, are accessible to the transaction via the current page table as shown in Figure 11.16. The old version of the changed pages remains unchanged and these pages continue to be accessible via the shadow page table.

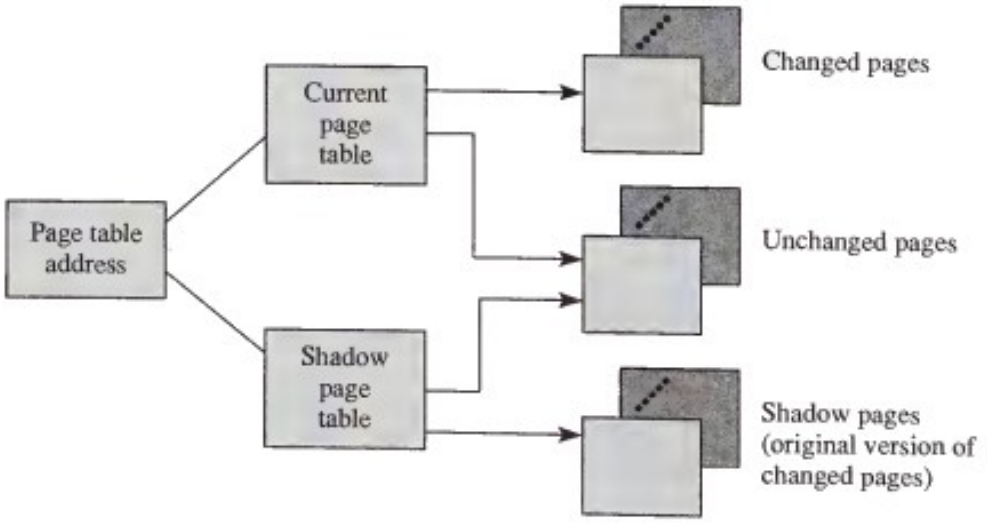

Figure 11.16 *Shadow Page Scheme*

The shadow page table contains the entries which existed in the page table before the start of the transaction and points to blocks that were never changed by the transaction. The shadow page table remains unaltered by the transaction and is used for undoing the transaction.

Now let us see how the transaction accesses data during the time it is active. The transaction uses the current page table to access the database blocks for retrieval. Any modification made by the transaction involves a write operation to the database and the shadow page scheme handles the first write operation to a given page as follows:

- A free block of nonvolatile storage is located from the pool of free blocks accessible by the database system.
- Copying the block to be modified onto this block.
- The original entry in the current page table is changed to now point to this new block.
- The updates are propagated to the block pointed to by the current page table which, in this case, would be the newly created block.

Subsequent write operations to a page already duplicated is handled via the current page table. Any changes made to the database are propagated to the blocks pointed to by the current page table. Once a transaction commits, all modifications made by the transaction and still in buffers are propagated to the physical database (i.e. the changes are written on to the blocks pointed to by the current page table). The propagation is confirmed by adopting the current page table as the table containing the consistent database. The current page table or the active portion of it could be in volatile storage. In this case a commit transaction causes the current page table to be written out to nonvolatile storage.

In the case of a system crash before the transaction commits, the shadow page table and the corresponding blocks containing the old database, which was assumed to be in a consistent state, will continue to be accessible.

To recover from system crashes during the life of a transaction, all we have to do is revert to the shadow page table so that the database remains accessible after the crash. The only precaution to be taken

is to store the shadow page table onto stable storage and have a pointer, which points to the address where the shadow page table is stored, accessible to the database through any system crash.

Committing a transaction in the shadow page scheme requires that all the modifications made by the transaction are propagated to physical storage, and the current page table be copied onto stable storage. Having so done, the shadow page scheme reduces the problem of propagating a set of modified blocks to the database, to that of changing a single pointer value contained in the page table address from the shadow page table address to the current page table address. This can be done in an atomic manner and is not interrupt-able by a system crash.

In the case of a system crash occurring any time between the start of a transaction and the last atomic step of modifying a single pointer from the shadow page to the current page, the old consistent database is accessible via the shadow page table and there is no need to undo a transaction. A system crash occurring, after the last mentioned atomic operation, will have no effect on the propagation of the changes made by the transaction; these changes will be preserved and there is no need for a redo operation.

The shadow blocks (i.e. the old version of the changed blocks) can be returned to the pool of available nonvolatile storage blocks to be used for further transactions.

The undo operation in the shadow page scheme consists of discarding the current page table and returning the changed blocks to a pool of available blocks.

The advantages of the shadow page scheme is that the recovery from system crash using this scheme is relatively inexpensive and this is achieved without the overhead of logging.

Before we go on to another method of indirect update it is worth mentioning some of the drawbacks of the shadow page scheme. One of the main disadvantages of the shadow scheme is the problem of scattering. This problem is critical in database systems because of the fact that over a period of time the database will be scattered over the physical memory and related records may require a very large access time. For example, two records which are required together and originally placed in blocks on the same cylinder of a disk may end up on the extreme cylinders on that same disk. Accessing these records together now, will involve moving the read/write head over the entire surface of the disk and, hence, a large access time.

The other problem with the shadow page scheme was already mentioned: when a transaction commits, the original version of the changed blocks pointed to by the shadow page table have to be returned to the pool of free blocks, otherwise such pages will become inaccessible. If this is not done successfully, when a transaction commits (perhaps due to a system crash), such blocks become inaccessible and require a ***garbage collection*** operation to be performed periodically to reclaim such lost blocks.

Shadow paging for concurrent transactions requires additional bookkeeping and in such an environment some logging scheme is used as well.

Reflecting Updates to the Database via Logs and Recovery

In the update via logs scheme, the transaction is generally not allowed to modify the database. All changes to the database are deferred until the transaction commits. However, as in the update in place scheme, all modifications made by the transaction are logged. Furthermore, since the database is not

modified directly by the transaction, the old values are not required to be saved in the log. Once the transaction commits, the log is used to propagate the modifications to the database.

During the life of a transaction, all output operations to the database are intercepted which causes an entry to be made in the log for the transaction. This entry contains the identification of the items being updated, along with the new values. When the transaction starts a commit operation, a commit transaction mark is written onto the log. After this step, the log is used to modify the database.

A system crash, occurring during the time when a transaction is active, does not require an undo operation since the database was not directly changed by the transaction. A system crash, occurring after the transaction commits, can be recovered from the log maintained for the transaction.

Let us return to the example of transferring a part from inventory to a project given in the program of Figure 11.12(a). Figure 11.17 gives the log for the transactions corresponding to the transfer of 100 units of part $Part_1$ from inventory to project $Proj_5$, followed by a transaction corresponding to the transfer of 10 units of part $Part_4$ from inventory to project $Proj_2$. The log contains only redo information and the only operations performed during the life of a transaction on the physical database is that of reads.

Now, let us assume various scenarios for a system crash. First, consider a system crash which occurs any time before the step s_7; this step corresponds to the writing of the commit transactions T_0 step. This system crash will require the recovery system to undo the effect of transaction T_0, which in this case involves discarding the log for transaction T_0, which lacks the Commit transaction marker. The values for the record corresponding to $Part_1$ and $Proj_5$ had not been propagated to the database.

If the system crash occurs after the completion of step s_7, then, when the system is restarted,the recovery system will find the commit transaction marker for transaction T_0. It will then redo the transaction to ensure that the effects of the transaction T_0 are correctly propagated to the database. The redo operation needs only the new values for the fields modified by the transaction in the records for $Part_1$ and $Proj_5$. After the redo operation, the database is restored to the state existing at the end of the transaction T_0.

A crash occurring during the recovery operation will not effect the subsequent recovery operation, since the redo operation is idempotent.

A crash occurring after the step s_{17} in Figure 11.17 requires the recovery system to redo both transactions T_0 and T_1.

The recovery system, after a system crash, checks the log. For those transactions that contain both a start transaction marker and an end transaction marker, it will initiate a redo transaction operation. A partially complete transaction in the system log is indicated by a start of transaction marker without, a corresponding end of transaction marker. Such partially complete transactions are ignored by the recovery system since they will not have modified the database.

However, we must distinguish an update made by a partially complete transaction from a partial update made from the log of a committed transaction in the deferred update from log phase. A partially completed update (updated during the end of transaction processing after a commit transaction is executed by the program controlling the transaction) cannot be undone with the deferred update using the log scheme; it can only be completed or redone. The only way it can be undone is by a compensating transaction to undo its effects (as is the case in standard accounting practice).

Step	Transaction Action	Log Operation	Database Operation
s_0	Start of T_0	**Write**(start Transaction T_0)	
s_1	*get*($Part_1$)		**Read**($Part_1$)
s_2	modify(Q_in_S from 400 to 300)		
s_3	*put* ($Part_1$)	**Write**(record for Part#=$Part_1$, new value of Q_in_S:300	
s_4	*get*($Proj_5$)		**Read**($Proj_5$)
s_5	modify(Q_to_D from 100 to 200)		
s_6	*put*($Proj_5$)	**Write**(record for Project#=$Proj_5$, new value of Q_to_D:200)	
s_7	Start Commit	**Write**(Commit transaction T_0);	
s_8	Commit/End of T_0		**Write**($Part_1$, $Proj_5$);
s_9	Start of T_1	**Write**(start Transaction T_1)	
s_{10}	*get*($Part_4$)		**Read**($Part_4$)
s_{11}	modify(Q_in_S from 600 to 590)		
s_{12}	*put* ($Part_4$)	**Write**(record Part#=$Part_4$, new value of Q_in_S:590)	
s_{13}	*get*($Proj_2$)		**Read**($Proj_2$)
s_{14}	modify(Q_to_D from 50 to 60)		
s_{15}	*put*($Proj_2$)	**Write**(record Project#=$Proj_2$, new value of Q_to_D:60)	
s_{16}	Start Commit	**Write**(Commit transaction T_1);	
s_{17}	Commit/End of T_1		**Write**($Part_4$, $Proj_2$)

Figure 11.17 *Entries for Indirect Update Log*

11.5 Buffer Management, Virtual Memory, and Recovery

The input and output operation required by a program, including a DBMS application program, is usually performed by a component of the operating system and it normally uses **buffers** (reserved blocks of primary memory) to match the speed of the processor and the relatively fast primary memories with the slower secondary memories, and to minimize, whenever possible, the number of input and output operations between the secondary and primary memories. The assignment and management of memory block is called **buffer management** and the component of the O.S. that performs this task is usually called the **buffer manager**.

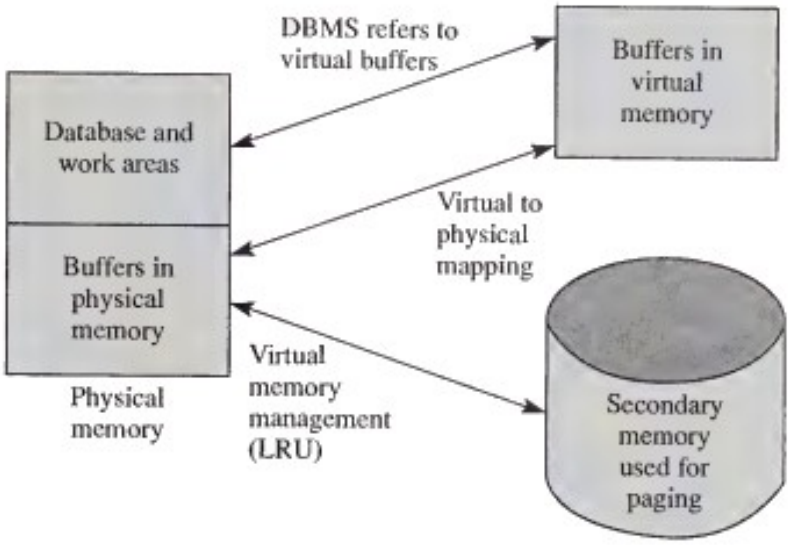

Figure 11.18 *DBMS Buffers in Virtual Memory*

The goal of the buffer manager is to ensure that as many as possible of the data requests made by programs are satisfied from data copied from secondary storage devices into the buffers. In effect, a program performs an input or an output operation using, let us say, *get* or *put* statement; the buffer manger will be called on to respond to these input or output request. It will check to see if the request for the data can be satisfied by reading from, or writing to, the existing buffers. If this is so, the input or output operation occurs between the program work area and buffers. If, for example, an input request cannot be so satisfied, then the buffer manager will have to do a physical transfer between the secondary memory and a free buffer, and then make the data, so placed in the buffer available, to the program requesting the original input operation. A similar scenario will take place in the reverse order for an output: the buffer manager making a new buffer available to the program performing a *put* operation. The buffer manager performs the physical transfer between the buffer and the secondary memory by means of, let us say, **Read** and **Write** operations, whenever there is an anticipated need for new buffers, and none are available in a pool of free buffers for the current program. For sequential processing, the buffer manager can provide higher performance by pre-fetching the next block of data, and by batching write operations unto the commit phase of a transaction.

We have assumed so far that the buffer manager uses buffers which are in physical memory. However, in a computer system which uses a virtual memory management scheme, the buffers are in effect virtual memory buffers; there being an additional mapping between a virtual memory buffer and the physical memory as shown in Figure 11.18. Since the physical memory is managed by the memory management component of the operating system, a virtual buffer inputted by the buffer manager, may have been paged out by the memory manager in case there is insufficient space in the physical memory.

In a virtual memory management scheme, the buffers containing pages of the database undergoing modification by a transaction could be written out to secondary storage; the timing of this premature writing back of a buffer is independent of the state of the transaction and will be decided by the replacement policy used by the **memory manager**, which again is a component of the operating system. Thus, the page replacement scheme is entirely independent of the database requirements; these

requirements being that records undergoing modifications by a partially completed transaction not be written back, and the records for a committed transaction be rewritten, especially in the case of the update in place scheme.

It has been found that the locality of reference property is applicable to database buffers and, hence, to decrease the number of buffer faults, the least recently used (LRU) algorithm is used for buffer replacement. However, the normal LRU algorithm is modified slightly, and each transaction is allowed to maintain a certain number of pages in the buffer.

The buffering scheme can be used in the recovery system, since it effectively provides a temporary copy of a database page to which modifications can be directed, and the original page can remain unchanged in the nonvolatile storage medium. Both the log and the data pages will be written onto the buffer pages in virtual memory. The commit transaction operation can be considered to be a two phase operation (called a ***two phase commit***): the first phase is when the log buffers are written out (write-ahead-log), and the data buffers are written in the second phase of the commit operation. In case the data page is being used by another transaction, the writing of that page can be delayed. This will not cause a problem since the log is always forced during the first phase of commit. With this scheme the UNDO log is not required, since no uncommitted modifications are reflected in the database which would have to be undone as a result of a transaction abort or a system crash before commit.

In sequential processing of the database, the buffer manager pre-fetches the database pages. However, pages of data, once used need not follow the locality property. A page, once accessed, is now less likely to be accessed again. Hence, the buffer manager can use a modified LRU replacement algorithm, by using not one but two LRU lists: one is for randomly accessed pages and the second one is for sequentially accessed pages. Buffers needed for sequential processing are obtained from the sequential LRU list (i.e. one of the sequential LRU page is replaced to make room for the incoming page of data), if this list is longer than some pre-established length; otherwise, the buffer is obtained from the LRU list.

Let us take the example of Figure 11.12(a), corresponding to the program for transferring specified quantities of parts from inventory to projects. If the memory manager is using a LRU page replacement scheme, then a committed transaction may not have its page written back long after it commits. The reason for this is that the program has many transactions, each needing different records, but these records may be clustered on the same physical block of secondary memory. A transaction committing may have used the same page as the page required by the next transaction. However, such a page will not be written back by the memory manager using the simple LRU page replacement scheme. This, in turn, means that an update made by a committed transaction would not be reflected in the physical database which would create havoc in the recovery scheme.

The write-ahead-log protocol assumes that the undo log information for a transaction will be written to stable storage before the modifications made by a transaction are reflected in the database, and the redo portion of the log is written before the transaction commits. Under the memory and buffer managers of the operating system, we cannot assume that the buffers containing the log information are written ahead of the changes made to the database.

What this means is that the buffer manager, at least for those buffers used by the DBMS and its application programs, be under control of the DBMS, and the DBMS enforces the correct writing out of the buffers assigned for the log and the data at an appropriate time. The terms **steal** and **force** are used to indicate the buffer control mechanism. Steal indicates that the modified pages of data in the buffers may be written to the database at any time (as in the case of update in place scheme) and **not steal** means that

the modified pages are kept in the buffer until the transaction commits. In the case of not steal buffer control, (wherein no changes are propagated to the database during the life of a transaction), we have to decide what is to be done when a transaction starts to commit. If during this end of transaction processing, all modifications are actually propagated to the database, then we are assuming that the buffers are being **forced**. If no such forced writing of the buffers can be assumed during the end of transaction processing, then the updates cannot be presumed to have been actually propagated to the database. This in turn requires that with the **no force** strategy, committed transactions have to be redone in the case of a system crash. In the case of forcing, no redone is required for committed transactions, the modifications made by the committed transactions can be safely assumed to have been propagated to the database.

11.6 Other Logging Schemes

In our discussions so far we have assumed that the logging scheme writes the following details in the log: the identification of the records being modified, the modified values of each record, and in some cases the old values of each record modified. This is the so called **record level logging**. However in addition to the record level logging, other schemes can be used. We describe below the record level logging, as well as the page level, and the DML level logging schemes.

RECORD LEVEL LOGGING: Here, instead of recording the entire page whenever a modification is done anywhere on a page, the log is kept of the before and the after image of the record that undergoes modification. Insertion of a new record can be handled by using null values for the before image, and deletion of an existing record is indicated by using null values for the after image. The advantage of this scheme is the obvious; the amount of space needed for the log is much lower**.**

PAGE LEVEL LOGGING: In this scheme, the entire page is recorded in the log, whenever a record within the page is modified; for the UNDO operation, the entire page before any modification is written to the log and for the REDO operation, the entire page after the modifications is written onto the log.

If a number of changes are made on the same page, a design decision has to be made regarding the number of page images that will be stored in the log. One choice is to have only one before image and one after image; the former being the image at the start of the transaction, the latter, that at the end of the transaction. Another alternative is to have one before and one after image for each change. (This requires that if there are n changes made on a page, there will be 2n page images, the page image number 2i and 2i+1, for 1 ¾ i ¾ n-1, being the same! The order of i, here, is a chronological order.)

In a modification of the page level logging scheme, instead of writing the before image of the page and the after image of the page to the log, a difference of these two, in the form of an exclusive or, is written in a compressed form to the log. Since only a few bytes of a page will be changed as a result of an update transaction on a record contained on the page, the exclusive or of the before and after image of the page will give a large number of zeros which can be compressed using an appropriate data compression method.

QUERY LANGUAGE LOGGING: In this approach the log entry of the data manipulation statements modifying the database, along with the parameters used by the statements, are recorded in the log. The parameters would include the record identifiers and values of attributes of the record being modified. As in the case of the record level logging, appropriate null values can be used for the records

being deleted. In case the update is made by a higher level language program, these updates can be reduced to statements that operate on a single record; the latter would be recorded along with the parameters in the log. The redo recovery function requires re-executing the logged data manipulation statements with their parameters. The undo recovery function requires generating reverse data manipulation statements corresponding to the logged statements and executing these reverse statements. To undo the effect of a DELETE statement requires the generation of an INSERT statement, and the parameter would be the identifier of the record to be inserted along with the before image of the record.

11.7 Cost Comparison

Let us briefly compare the cost of the various recovery schemes we discussed, namely the update in place, the deferred update with shadow page scheme, and the deferred update using log.

If an update in place scheme is used along with a buffer scheme where partially modified pages can be written at any time and all modified pages are written prior to a commit transaction, then the cost of an undo operation is relatively high, though, the cost of a redo is very low. In this case each end of transaction is a checkpoint, since all modifications are forced to be written to nonvolatile storage. However, if all the modified pages are not forced to be written during the end of transaction processing, then the cost of both an undo and a redo are relatively higher. Furthermore, the end of a transaction is not a checkpoint in this scheme.

If an update in place scheme is used along with a not steal and force buffer scheme where partially modified pages are not allowed to be written at any time, (the writing of such modified pages being delayed till the end of the transaction processing, and it is only at this point when all pages are written), then the costs of undo and redo is very low. Again each end of a transaction represents a checkpoint.

With an indirect update scheme, where the end of transaction forces all modified pages to be processed, the cost of the undo and redo are relatively lower.

If the database system defers the propagation of changes to the database until the commit operation, then in case the transaction is rolled back by the program controlling the transaction, the changes made by the transaction need not be rolled back. The rollback operation in this case consists of merely not propagating the modifications made by the transaction to the DBMS. The same procedure will apply if the system aborts the transaction.

11.8 Disaster Recovery

Disaster here is used to denote circumstances which result in the loss of the physical database stored on the nonvolatile storage medium. This implies that there will also be a loss of the volatile storage, and the only reliable data is the data stored in stable storage. The data stored in stable storage consists of the archival copy of the database and the archival log of the transactions on the database represented in the archival copy**.**

The recovery process requires a global redo. In a global redo the changes made by every transaction in the archival log are redone using the archival database as the initial version of the current database. The

order of redoing the operations must be the same as the original order, and, hence, the archival log must be chronologically ordered.

Since the archival database should be consistent, it must be a copy of the current database in a quiescent stage (i.e., no transaction can be allowed to run during the archiving process). The quiescent requirement dictates that the frequency of archiving be very low. The time required to archive a large database and the remote probability of a loss of nonvolatile storage exacerbate this avoidance of too many archiving with the net result that archiving is performed, let us say, at quarterly or monthly intervals.

The low frequency of archiving the database means that the number of transactions in the archival log will be large and this in turn leads to a lengthy recovery operation (of the order of days).

A method of reconciling the abhorrence for archiving and the heavy cost of infrequent archiving at the time of recovery, is to archive more often in an incremental manner. In effect, the database is archived in a quiescent mode very infrequently, but what is archived at more regular intervals is that portion of the database that was modified since the last incremental archiving. The archived copy can then be updated to the time of the incremental archiving without disrupting the on-line access of the database. This updating can be performed on an entirely different computer system.

The recovery operation consists of redoing the changes made by committed transactions from the archive log on the archive database. A new consistent archive database copy can be generated during this recovery process.

11.9 Summary

In this chapter we discussed the recovery of the data contained in a database system after failures of various types. The reliability problem of the database system is linked to the reliability of the computer system on which it runs. The types of failures, that the computer system is likely to be subject to, include that of components or subsystems, software failures, power outages, accidents, unforeseen situations, and natural or man-made disasters. Database recovery techniques are methods of making the database fault-tolerant; the aim of the recovery scheme is to allow database operations to be resumed after a failure with a minimum loss of information and at an economically justifiable cost.

In order that a database system works correctly, we need correct data, correct algorithms to manipulate the data, correct programs that implement these algorithms, and, of course, a computer system that functions accurately. Any source of errors in each of these components has to be identified, and a method of correcting and, recovering from ,these errors has to be designed in the system.

A ***transaction*** is a program unit whose execution may change the contents of the database. If the database was in a consistent state before a transaction, then on completion of the execution of the program unit, corresponding to the transaction, the database will be in a consistent state. This in turns requires that the transaction be considered atomic: it is executed successfully or, in case of errors, the user views the transaction as not having been executed at all.

A database recovery system is designed to recover from the following types of failures:

Failures without loss of data
Failure with loss of volatile storage
Failure with loss of nonvolatile storage

Failure with a loss of stable storage:

The basic technique to implement the database recovery is by using data redundancy in the form of **logs**, **checkpoints** and **archival** copies of the database.

The log contains the redundant data required to recover from volatile storage failures and also from errors discovered by the transaction or database system. For each transaction the following data is recorded on the log: the start-of-transaction marker, transaction identifier, record identifiers, the previous value(s) of the modified data, the updated values; and if the transaction is committed, then a commit transaction marker, otherwise, an abort or rollback transaction marker.

The checkpoint information is used to limit the amount of recovery operations to be done following a system crash resulting in the loss of volatile storage.

The archival database is the copy of the database at a given point in time stored onto stable storage. It contains the entire database in a quiescent mode and is made by simple dump routines to dump the physical database onto stable storage. The purpose of the archival database is to recover from failures that involve loss of nonvolatile storage. The archiving process is a relatively time-consuming operation, and during this period the database is not accessible. Consequently, archiving is done at very infrequent intervals. The archive log is used for recovery from failures involving loss of nonvolatile information. The log contains information on all transactions made on the database from the time of the archival copy; the log is written in a chronological order. The recovery from loss of nonvolatile storage uses the archival copy of the database and the archival log to reconstruct the physical database to the time of the nonvolatile storage failure.

Whenever a transaction is run against a database, a number of options can be used in reflecting the modifications made by the transactions. The options we have examined are:

-Update in place

-Indirect update with careful replacement: There are two possibilities which can be considered. These are the shadow page scheme and the update via logs scheme.

In the update in place scheme, the transaction updates the physical database and the modified record replaces the old record in the database. However, the write-ahead-log strategy is used, and the log information about the transaction modifications are written before update operations initiated by the transactions is performed.

The shadow page scheme uses two page tables for a transaction that is going to modify the database. The original page table is called the shadow page table, and the transaction addresses the database using another table called the current page table. Initially both page tables point to the same blocks of physical storage. The current page table entries may change during the life of the transaction. The changes are made whenever the transaction modifies the database by means of a write operation to the database. The pages that are affected by a transaction are copied onto new blocks of physical storage and these blocks, along with the blocks not modified, are accessible to the transaction via the current page table. The old version of the changed pages remains unchanged, and these pages continue to be accessible via the shadow page table. In the shadow page scheme, propagating a set of modified blocks to the database is achieved by changing a single pointer value contained in the page table address from the shadow page table address to the current page table address. This can be done in an atomic manner and is uninterruptible by a system crash.

In the update via logs scheme, the transaction is not allowed to modify the database. All changes to the database are deferred until the transaction commits. However, as in the update in place scheme, all modification made by the transaction are logged. However, since the database is not modified directly by the transaction, the old values are not required to be saved in the log. Once the transaction commits, the log is used to propagate the modifications to the database.

The recovery process from a failure resulting in the loss of nonvolatile storage requires a global redo, i.e., redoing the effect of each and every transaction in the archival log, the archival database being used as the initial version of the current database. The order of performing an undo or a redo operation must be the same as the original order, and, hence, the archival log file must be chronologically ordered.

Key Terms

action consistent check-pointing
archival database
archival database
archival log
atomic operation
atomicity
audit trails
availability
backward error recovery
buffer management
careful replacement
checkpoints
commit
compensating transaction
consistency
consistency error
current database
current log
current page table
design errors
disaster recovery
durability
error
external failure
failure
fault
fault-tolerant
forced
forward error recovery
garbage collection
global redo
global undo
indirect update
isolation
journal
logs
materialized database
mean time between failures
mean time to repair
no force
nonvolatile storage
not steal
over utilization and overloading
page level logging
physical database
poor quality control
query language logging
record level logging
redundancy
reliability
rollback
shadow page scheme
shadow page table
stable storage
steal
system error
transaction consistent check-pointing
transaction do
transaction idempotent
transaction oriented check-pointing
transaction redo
transaction undo
transactions
two phase commit
update in place
update via log
user error
virtual memory
volatile storage
wear-out

least recently used write-ahead log strategy

Exercise

11.1 What if any thing can be done to recover the modifications done by partially completed transactions that are running at the time of a system crash? Can on-line transactions be so recovered?

11.2 In a database system that uses an update in place scheme, how can the recovery system recover from a system crash if the write ahead protocol is used for the log information.

11.3 What modifications have to be done to a recovery scheme if the transactions are nested? (A nested transaction is a transaction where one transaction is contained within another transaction.)

11.4 n the recovery technique known as forward error recovery, on the detection of a particular error in a system, the recovery procedure consists of adjusting the state of the system to recover from the error (without suffering the loss that could have occurred because of the error). Can such a technique be used in a DBMS system to recover from system crashes with the loss of volatile storage?

11.5 Show how the backward error recovery technique is applied to a DBMS system which uses the update in place scheme to recover from a system crash with a minimum loss of processing.

11.6 If the checkpoint frequency is too low, then a system crash will lead to the loss of a very large number of transactions and a very long recovery operation; if the checkpoint frequency is very high, then the cost of check-pointing is very high. Can you suggest a method of reducing the frequency of check-pointing without incurring a heavy recovery operation and at the same time reducing the number of lost transactions?

11.7 How can a recovery system deal with recovery of interactive transactions on on-line systems such as banking, or airline reservation? Suggest a method which can be used, in such systems, to restart active transactions after a system crash.

11.8 For a logging scheme based on DML, give the kind of log entry required, and indicate the UNDO and the REDO part of the log.

11.9 If the write-ahead-log scheme is being used, compare the strategy of writing the partial update made by a transaction, to the database, to the strategy of delaying all writes to the database till the commit.

11.10 How is the checkpoint information used in the recovery operation following a system crash?

11.11 Define the following terms:

- Write-ahead-log strategy
- ransaction consistent checkpoint
- Action consistent checkpoint
- Transaction oriented checkpoint
- Two-phase commit

11.12 Compare the shadow page scheme with the update in place with forced and no steal buffering from the point of view of recovery.

11.13 Explain why no undo operations need be done for recovery from loss of nonvolatile storage loss.

11.14 What type of software errors can cause a failure with loss of volatile storage?

11.15 What is the difference between transaction oriented check-pointing and the write-ahead-log strategy?

11.16 What are the advantages and disadvantages of each of the methods of logging discussed in Section 11.6?

11.17 Consider the update-in-place scheme, where the database system defers the propagation of updates to the database until the transaction commits(see 11.4.1). Describe the recovery operations that have to be undertaken following a system crash with loss of volatile storage.

Bibliographic Notes

Some of the earliest works in recovery were reported in [Oppe68], [Chan72], [Bjor73], and [Davi73]. Analytical models for recovery and rollback and discussions on these are presented in [Chan75]. The concept of transaction and its management is presented in [Gray78]. The recovery system for System R is presented in [Gray81]. The shadow page scheme used in system R is described in an earlier paper [Lori77]. [Verh78] is an early survey article on database recovery; [Haer83],and [Kohl81] are more recent survey articles based on the transaction paradigm. An efficient logging scheme for the UNDO operation is discussed in [Reut80]. [Teng84] discusses the buffer management function to optimize database performance for the DB2 relational database system.

The concept of nested transaction was discussed by [Gray81]; more recent discussions are presented in [Moss85].

Textbooks discussing the recovery operation are [Bern88], [Date83], [Date86], and [Kort86]. Reliability concepts are presented in [Wied83]

Bibliography

[Bern88] Bernstein P., Hadzilacos, V., Goodman, N., Concurrency Control and Recovery in Database System, Addison Wesley, Reading, MA, 1988.
[Bjor73] Bjork, L.A. "Recovery Scenario for a DB/DC System", Proc. of the ACM Annual Conference, 1973, pp142-146.
[Chan72] Chandy, K.M., Ramamoorthy, C.V., "Rollback and Recovery Strategies for Computer Programs", IEEE, Vol. C-21-6, June 1972, pp546-555.
[Chan75] Chandy, K.M., Browne, J. C., Dissly, C.W., Uhrig, W. R. "Analytic Models for Rollback and Recovery Strategies in Data Base Systems", IEEE, Vol. SE-1-1, March 1975, pp100-110.
[Date83] Date, C.J., "An Introduction to Database Systems", Vol. 2, fourth edition, Addison Wesley, Reading, MA, 1983
[Date86] Date, C.J., "An Introduction to Database Systems", Vol. 1, Addison Wesley, Reading, MA, 1986
[Davi73] Davies Jr., J.C., "Recovery Semantics for a DB/DC System", Proc. of the ACM Annual Conference, 1973, pp136-141.
[Gior76] Giordano, N.J., Schwartz. M.S., "Database Recovery at CMIC", Proc ACM SIGMOD Conf. on Management of Data, June 1976, pp33-42.
[Gray78] Gray, J.N. "Notes on a Database Operating Systems", in Operating Systems: An Advanced Course, Ed. R. Bayer et. al., Springer-Verlag, Berlin, 1978.
[Gray81] Gray, J.N. "The Transaction Concept: Virtues and Limitations", Proc. of the Intnl. Conf. on

VLDB, 1981, pp 144-154.
[Gray81a] Gray, J.N., McJones, P., Blasgen, M., Lindsay,B., Lorie, R., Price, T., Putzolu, F., Traiger, I., "The Recovery Manager of the System R Database Manager", ACM Computing Surveys, Vol. 13-2, June 1981, pp 223-242.
[Haer83] Haerder, T., Reuter, A., "Principles of Transaction_Oriented Database Recovery", ACM Computing Surveys, Vol. 15-4, December 1983, pp 287-317.
[Kohl81] Kohler, K. H., "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems",ACM Computing Surveys, Vol. 13-2, June 1981, pp 148-183.
[Kort86] Korth, H.F., Silberschatz, A., "Database System Concepts", McGraw Hill,New York, NY, 1986.
[Lori77] Lorie,R., "Physical Integrity in a Large Segmented Database", ACM TODS, Vol 2-1, March 1977, pp91-104.
[Lync83] Lynch,N.A., "Multilevel Atomicity- A New Correctness Criterion for Database Concurrency Control", ACM TODS, Vol8-4, December 1983, pp 484-502.
[Moss85] Moss, J. Eliot B., "Nested Transactions: An Approach to Reliable Distributed Computing", The MIT Press, Cambridge, MA., 1985.
[Oppe68] Oppenheimer, G., Clancy, K.P., "Considerations of Software Protection and Recovery from Hardware Failures", FJCC, AFIPS, Washington, D. C., 1968.
[Reut80] Reuter, A., "A Fast Transaction-Oriented Logging Scheme For UNDO Recovery", IEEE, Vol. SE6-4, July 1980, pp348-356.
[Seve76] Severance, D.G., Lohman, G.M., "Differential Files: Their Application to the Maintenance of Large Databases", ACM TODS, Vol 1-3, September 1976, pp 256-267.
[Teng84] Teng, J.Z., Gumaer, R.A., "Managing IBM Database 2 buffers to maximize performance", IBM Systems Journal, Vol.23-2, pp211-218, 1984.
[Verh78] Verhofstad, J. S. M., "Recovery Techniques for Database Systems", ACM Computing Surveys, Vol. 10-2, June, 1978, pp 167-195.
[Wied83] Wiederhold, Gio, "Database Design", second edition, McGraw Hill, New York, NY, 1983

Notes