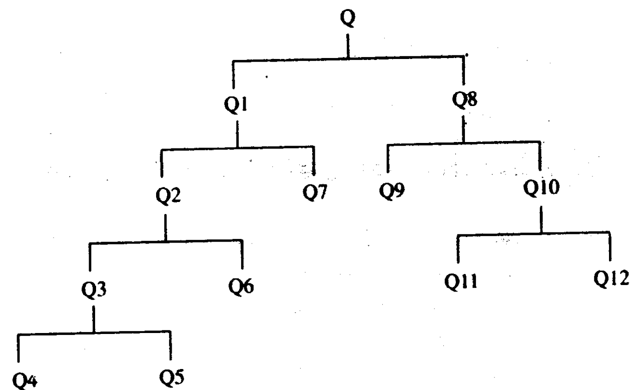
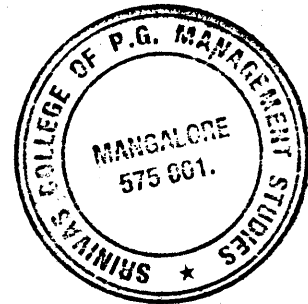
Figure 10.6 Decomposition of a query in the form of a tree.




SRINIVAS COLLEGE OF P.G. MANAGEMENT STUDIES
MANGALORE
575 001.

tution is to be performed. The objective of the optimization is to minimize the estimated costs.

Access Aids in N-Variable Expressions

The presence of access aids and the commonality of attributes can be used to advantage in the evaluations of multiple variable queries. Let us consider, for instance, the three-variable query, $U = R \bowtie S \bowtie T$. We can create indexes on the joining attributes in the join $R \bowtie S$ for R and $S \bowtie T$ for T, if they do not already exist. If these indexes have to be created, access to the relations R and T is involved, plus the cost of writing the indexes to secondary storage if insufficient space exists in main memory. Subsequently, the tuples of S are accessed. For each tuple of S, the required tuples from R and T are determined by using the values of the joining attribute and the indexes for R and T. In this manner, the three-way join could be evaluated.

The cost of this method is that of access to tuples of relation S and the required tuples of R and T, plus the cost of accessing the indexes. If the index must be created, the cost also entails the overhead of creating the indexes, plus access to each of the three relations followed by the selected tuples from the relations R and T.

10.8.4 Access Plan

Once the method of evaluating various operations is determined, the steps involved in combining the query components to deduce the final results have to be planned. Generating an optimal access plan is a stepwise process done in conjunction with the query transformation operation. In generating an access plan a decision has to be made regarding which indexes should be generated and which of the existing data structures should be used.

10.4 Repeat Exercise 4.12 from Chapter 4, giving both an optimal relational algebra expression and the corresponding query tree.

10.5 Consider the computation of the join R(*A,B,C*) and S(*B,C,D,*). Suppose R has 1,000 tuples stored 30 tuples per disk block and S has 10,000 tuples stored 40 tuples per disk block. There is space in the main memory for 3 buffers for relation R and 5 buffers for relation S. What is the number of disk accesses made if the relations are joined using the nested loop method?

10.6 Indicate if each of the following equivalences are valid, without any knowledge about the relation schemes of R and S. If valid, how could they be used in query modification to improve its evaluation?

(a) $\sigma_P(R - S) \equiv \sigma_P R - \sigma_P S$

(b) $\pi_P(R - S) \equiv \equiv \pi_P R - \pi_P S$

10.7 Given R(*A,B,C*), S(*B,C,D*), and T(*C,D,E*), draw the query tree for each of the following queries and apply optimization procedures to it.

(a) $\sigma_{B=b}(\pi_{ABC}(R \bowtie S) \cap \pi_{ABC}(R \bowtie T))$

(b) $\pi_{ABC}(\sigma_{B=b}(\pi_{AB}R) \bowtie \pi_{AB}S) - \pi_{ABC}(\sigma_{D=d}(R \bowtie T)))$

10.8 Consider the following query on the database discussed in this chapter.

```
select S.Std#, S.Std_Name
from STUDENT s,Grade g,Registration r,COURSE c,COURSE cl
where s.Std# = g.Std# and
    g.Course# = c.Course# and
    c.Course_Name = 'Database' and
    g.Grade = A and
    S.Std# = r.Std# and
    cl.Course# = r.Course# and
    cl.Course_Name = 'Database Design'
```

Assuming that the size of the relations are as indicated in the text, find the best strategy to evaluate this query.

10.9 Generate an optimal query tree for each query of Exercise 5.10 of Chapter 5.

10.10 Is it possible to use algebraic modification to convert the first relational algebraic version of the query in Section 10.2 to the third version? If so, depict a sequence of query trees showing each step of the modification process.

10.11 Consider the different access strategies (indexing and hashing). State how the availability of such access aids influences query processing.

10.12 Modify the algorithm for nested joins using block access wherein the join condition involves more than one attribute from each relation.

Bibliographic Notes

Wong and Youssefi (Wong 76) introduced the decomposition technique, Selinger et al. (Seli 79) describe access path selection, and Kim (Kim 82) describes join evaluation strategies. Techniques for query improvement are presented in Hall (Hall 76). Some join minimization techniques are presented in the textbooks by Maier (Maie 83) and Ullman (Ullm 82). Query evaluation algorithms are presented in Blasgen and Eswaren (Blas 77) and Yao (Yao 79). Join indexes for a two-variable join are presented in Valduriez (Vald 87). When two or more

relations are to be joined, the use of a composite B-tree–based index has been shown to be advantageous (Desa, in press, Desa 89). A survey of query processing techniques is given by Jarke and Kock (Jark 84). The distributed query processing survey by Yu and Chang (Yu 84) also considers techniques useful in centralized database systems.

Bibliography

(Blas 77) M. W. Blasgen & K. P. Eswaren, "Storage and Access in Relational Databases," *IBM Systems Journal* 16, 1977.

(Desa 89) B. C. Desai, "Performance of a Composite Attribute and Join Index," *IEEE Trans. on Software Engineering* 15(2), February 1989, pp. 142–152.

(Desa) B. C. Desai, F. Sadri, & P. Goyal, "Composite B-tree: An Access Aid for Query Processing and Integrity Enforcement," *Computer Journal* (in press).

(Hall 76) P. A. Hall, "Optimization of a Single Relational Expression in a Relational Database System," *IBM Journal of Research and Development* 20, pp. 244–257.

(Jark 84) M. Jarke & J. Koch, "Query Optimization in Database Systems," *ACM Computing Surveys* (162), 1984, pp. 111–152.

(Kim 82) W. Kim, "On Optimizing SQL-Like Nested Query," *ACM Transactions on Database Systems* 3, pp. 443–469.

(Maie 83) D. Maier, *Theory of Relational Databases*. Rockville, MD: Computer Science Press, 1983.

(Seli 79) P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, & T. G. Price, "Access Path Selection in a Relational Database Management System," Proceedings ACM SIGMOD Intl. Conf. on Management of Data, 1979, pp. 23–34.

(Ullm 82) J. D. Ullman, *Principles of Database Systems*. Rockville, MD: Computer Science Press, 1982.

(Vald 87) P. Valduriez, "Join Indices," *ACM Transactions on Database Systems* 12(2), June 1987, pp. 218–246.

(Wong 76) E. Wong, & K. Youssefi, "Decomposition—A Strategy for Query Processing," *ACM Transactions on Database Systems* 1(3), 1976, pp. 223–241.

(Yao 79) S. B. Yao, "Optimization of Query Evaluation Algorithms," *ACM Transactions on Database Systems* 4(2), 1979, pp. 133–155.

(Yu 84) C. T. Yu & C. C. Chang, "Distributed Query Processing," *ACM Computing Surveys* 16(4), December 1984.

Chapter

11

Recovery

Contents

11.1 Reliability
- 11.1.1 Types of Failures
  - *Hardware Failure*
  - *Software Failure*
  - *Storage Medium Failure*
  - *Implementation of Stable Storage*
- 11.1.2 Types of Errors in Database Systems and Possible Detection Schemes
- 11.1.3 Audit Trails
- 11.1.4 Recovery Schemes

11.2 Transactions
- 11.2.1 States of a Transaction
- 11.2.2 Properties of a Transaction
- 11.2.3 Failure Anticipation and Recovery

11.3 Recovery in a Centralized DBMS
- 11.3.1 Logs
- 11.3.2 Checkpoints
- 11.3.3 Archival Database and Implementation of the Storage Hierarchy of a Database System
- 11.3.4 Do, Undo, and Redo
  - *Transaction Undo*
  - *Transaction Redo*
  - *Global Undo*
  - *Global Redo*

11.4 Reflecting Updates onto the Database and Recovery
- 11.4.1 Update in Place
- 11.4.2 Indirect Update and Careful Replacement
  - *Reflecting Updates to the Database via Shadow Page Scheme and Recovery*
  - *Reflecting Updates to the Database via Logs and Recovery*

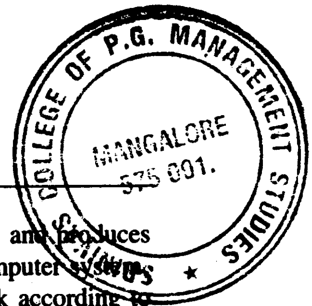
11.5 Buffer Management, Virtual Memory, and Recovery

11.6 Other Logging Schemes

11.7 Cost Comparison

11.8 Disaster Recovery

A computer system is an electromechanical device subject to failures of various types. The reliability of the database system is linked to the reliability of the computer system on which it runs. In this chapter we discuss 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; recovery issues of a distributed system are presented in chapter 15.


COLLEGE OF P.G. MANAGEMENT STUDIES
MANGALORE
575 001.

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 attained when all components of the system work according to specifications. The **failure** of a system occurs 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 judgment. The component in question is said to be in an erroneous state and further use of the component will lead to a failure that cannot be attributed to any other factor. A **fault** is detected either when 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. If an error is not propagated or perceived by another component of a system or by an user, it 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, the system returns the response that it was unable to retrieve the required information; furthermore, the system reports to a system error log that the error occurred and that it was a parity error. 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. We 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) to work 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 too low

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.

Overutilization and overloading: 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.

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

Storage Medium Failure

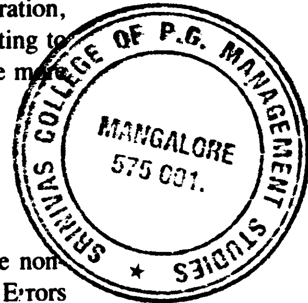
Storage media can be classified as volatile, nonvolatile, and permanent or stable.

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 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 an 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; 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 can experience a mechanical failure in the form of a **read/write head crash** (i.e., the read/write head comes 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 that cause the 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. 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.

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 the **archive database** and such archive data is usually stored in stable storage. Recovery data that 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 natural or man-made disasters. 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.


SRINIVAS COLLEGE OF P.G. MANAGEMENT STUDIES
MANGALORE
575 001.

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 write to the stable storage consists of writing the same block of data from volatile storage to distinct nonvolatile storage devices two or more times. If the writing of the block is done successfully, all copies of data will be identical and there will be no problems. If one or more errors are introduced in one or more copies, 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, the correct data is assumed to be the set that 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 failure, or external environmental conditions.

User error: This includes errors in application programs as well as errors made by online users of the database. One remedy is to allow online users limited access rights to the database, for example, read only. Any insertion or update operations require that appropriate validation check routines be 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 should be made here. **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 to the current values of other data-items in the database.

System error: This encompasses errors in the database system or the operating system, including situations such as **deadlocks** (see Section 12.8). Such errors are fairly hard to detect and require reprogramming the erroneous components of the

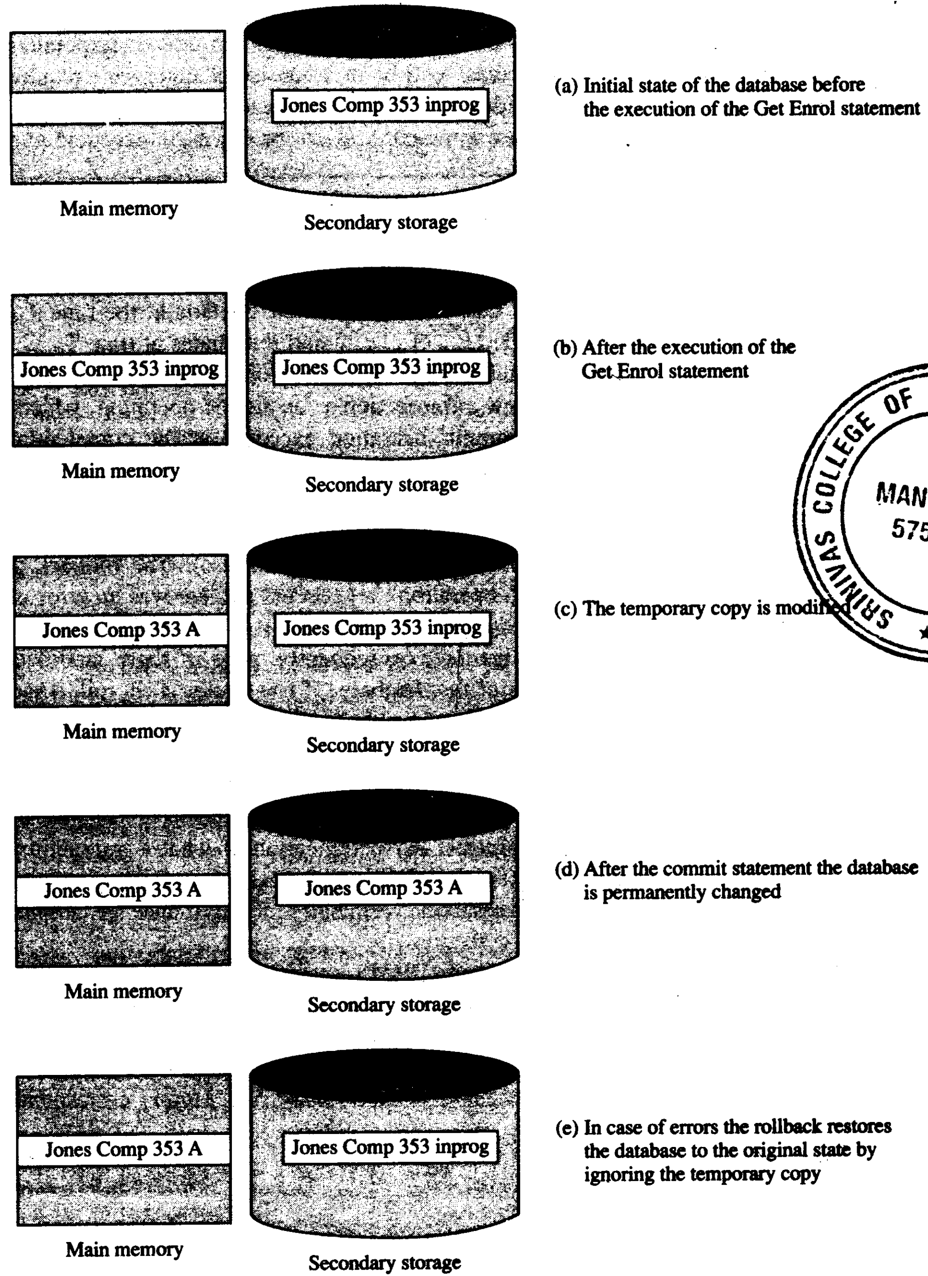
```
Procedure Modify_Enrol (Student_Name, Course, New_Grade);
  define action update ENROL(Student_Name, Course, Grade)as
     {* action update ENROL is defined as the next two
          statements *}
  begin
     get for update ENROL where
          ENROL.Student_Name = Student_Name and
          ENROL.Course = 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;
```

In this program the comment indicates the definition of the action **update** 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 during the execution of the statements defined for the action **update** ENROL. If there were any errors, we want to undo any changes made to the database by the statements defined for the update action. This involves 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. If there were no errors, we want the changes made by the update operations to become permanent bv being reflected in the actual database.

Figure 11.1 shows the successive states of the database system at different points of the execution of this program, with the change of student Jones's grade in course Comp353 from **in progress** to **A,** as shown in part d of the figure. In case there are any errors by the program, it ignores any modifications and the record for Jones remains unchanged as shown in part e.

The program unit Modify_Enrol given above consists of a number of statements, each of which is executed one at a time (each of the statements is compiled into a number of machine instructions, which are executed one at a time, sequentially). 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, to preserve 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.

Figure 11.1 Database states for program of Section 11.2.



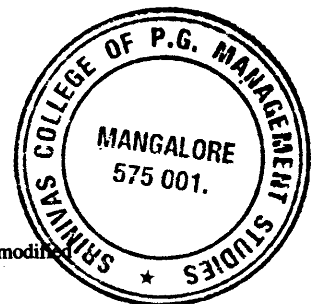
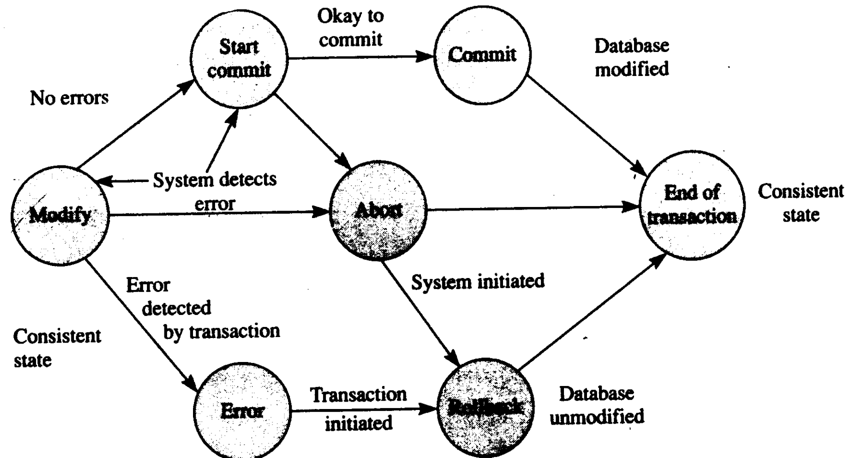

SRINIVAS COLLEGE OF P.G. MANAGEMENT STUDIES
MANGALORE 575 001.

Figure 11.3 Transaction states.



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

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, when it modifies the database. At the end of the modify state, there is a transition into one of the following states: start to commit, abort, or error. If 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 the buffers may or may not have been propagated to the database on the nonvolatile storage.

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 detects an error while in the modify state, it decides to terminate itself (suicide) and enters the error state and then, 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 that made no changes to the database is terminated without the need for a rollback, hence there are two paths in Figure 11.3 from the abort state to the end of the transaction. A transaction that, on the execution of its last statement, enters the start to commit state 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 11.3. 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 the 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 until the transaction terminates. Any notification of what a transaction is doing must not be communicated, for in stance via a message 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, represent the transaction paradigm.

The **atomicity** property of a transaction implies that it will run to completion as an indivisible unit, at the end of which either no changes have occurred to the database or the database has 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 consistent state.

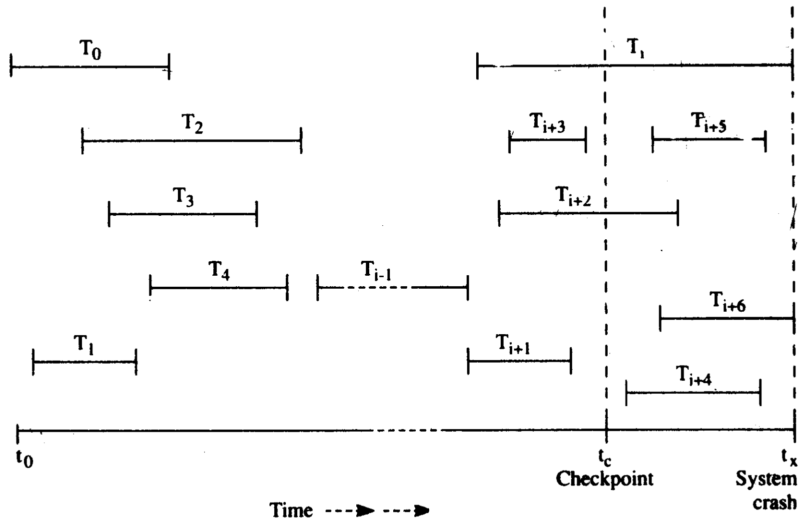
The **isolation** property of a transaction indicates that actions performed by a transaction will be isolated or hidden from outside the transaction until the transaction terminates. This property gives the transaction a measure of relative independence.

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 will not cause loss of the updates made by the transaction.

11.2.3 Failure Anticipation and Recovery

In designing a reliable system we try to anticipate different types of failures and provide for the means to recover without loss of information. Some very rare failures may not be catered to for economic reasons. Recovery from failures that are not thought of, overlooked, or ignored may not be possible. In common practice, the

Figure 11.5 Checkpointing.



Let us now see how the system can perform a recovery at time t_x. Suppose all transactions that started before the checkpoint time but were 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 to be undone. The undo list for the transactions of Figure 11.5 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, that transaction is removed from the undo list and placed in the **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 5 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 were 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 T_i and T_{i+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 and 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 the correct state of system files recorded explicitly in a backup file and 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.

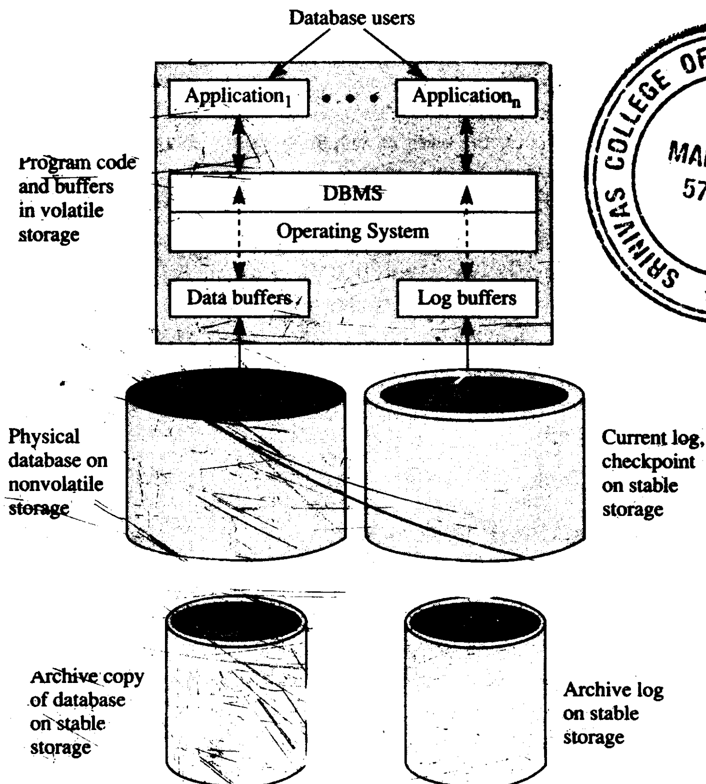
11.3.3 Archival Database and Implementation of the Storage Hierarchy of a Database System

Figure 11.6 gives the different categories of data used in a database system. These storage types are sometimes called the storage hierarchy. It consists of the archival database, physical database, archival log, and current log.

Physical database: This is the online copy of the database that is stored in nonvolatile storage and 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.

Figure 11.6 Database storage hierarchy.



SRINIVAS COLLEGE OF P.G. MANAGEMENT STUDIES
MANGALORE 575 001.

Transaction Redo

Transaction redo involves performing the changes made by a transaction that 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.

Global Undo

Transactions that 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 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 large. The log for the committed transactions needed for performing a global redo operation has to be stored on stable storage so that they are not lost with the loss of nonvolatile storage containing the physical database

11.4 Reflecting Updates to 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 the 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, in the 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. Two possibilities can be considered. The first scheme, called the **shadow page scheme,** makes the changes on a copy of that portion of the database being modified. The other scheme is called **update via log.** 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 examine these update schemes in greater detail.

SRINIVAS COLLEGE OF P.G. MANAGEMENT STUDIES MANGALORE 575 001.

11.4.1 Update in Place

In this scheme (see Figure 11.8) the transaction updates the physical database and the modified record replaces the old record in the database on nonvolatile storage. 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:

1. 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.
2. 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 11.9. 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 consists of the record(s) being modified, old values of the data items in the case of an update, and the 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

Figure 11.8 Update in place scheme.

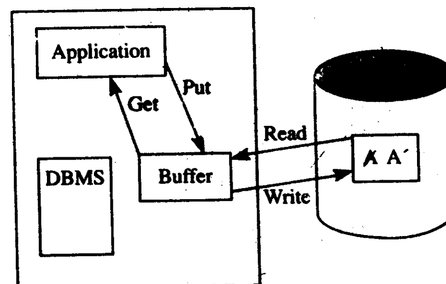
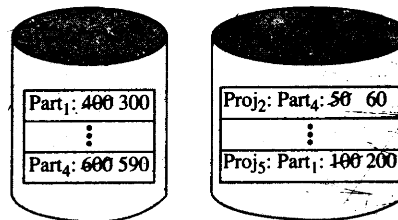


Figure 11.10 Modifications with update-in-place scheme.



gated 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 10. The program above 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 11.11. The first operation is called transaction T_0; the second operation, T_1. *Quantity_in_Stock* is abbreviated as *Q_in_S* and *Quantity_to_Date* as *Q_to_D*

Now suppose that while the program above 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, 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 restore 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, the recovery system will find an end-of-transaction marker for transaction T_0 in the log. The log entry will be as given below:

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

Figure 11.11 The steps for two transactions.

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		

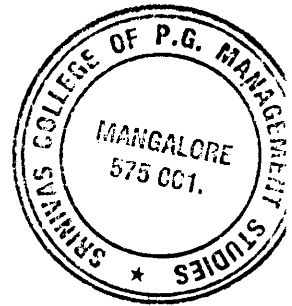
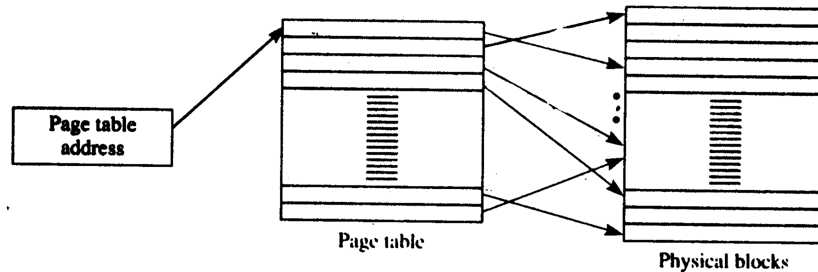


Figure 11.13 Paging scheme.

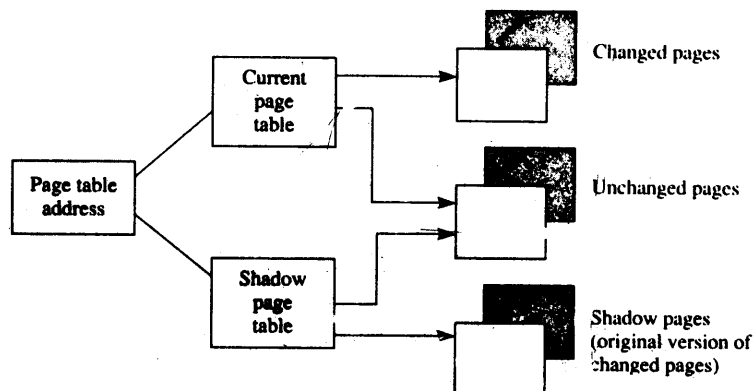


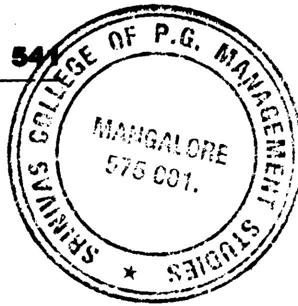
shadow page table and the transaction addresses the database using another page table known as the **current page table.** Initially, both page tables point to the same blocks of physical storage. The current page table entries may change during the life of the transaction. The changes are made whenever the transaction modifies the database by means of a write operation. The pages that are affected by a transaction are copied 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 11.14. The old version of the changes pages remains unchanged and these pages continue to be accessible via the shadow page table.

The shadow page table contains the entries that 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. The shadow page scheme handles the first write operation to a given page as follows:

Figure 11.14 Shadow page scheme.





- A free block or nonvolatile storage is located from the pool of free blocks accessible by the database system.
- The block to be modified is copied onto this block.
- The original entry in the current page table is changed to 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 are 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 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 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 on stable storage and have a pointer that points to the address where the shadow page table is stored and that is 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 be propagated to physical storage and the current page table be copied to stable storage. Then 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 interrupted 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 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 advantage of the shadow page scheme is that the recovery from system crash 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 data-

log is indicated by a start transaction marker without a corresponding end transaction marker. Such partially complete transactions are ignored by the recovery system since they will not have modified the database.

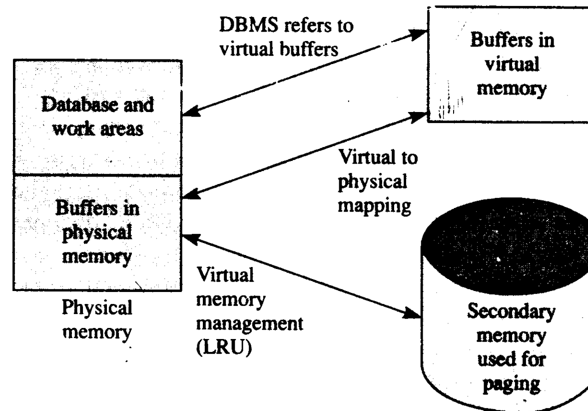
However, we must distinguish an update made by a partially complete transaction from a partial update made from the log of a committed transaction in the deferred update from the log phase. A partially completed update (updated during the end of transaction processing after a commit transaction is executed 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).

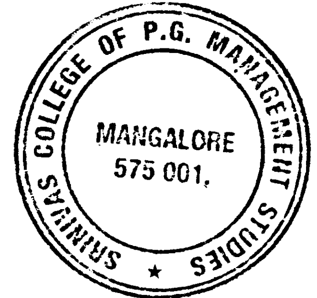
11.5 Buffer Management, Virtual Memory, and Recovery

The input and output operations required by a program, including a DBMS application program, are usually performed by a component of the operating system. These operations normally use 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 blocks is called **buffer management** and the component of the operating system that performs this task is usually called the **buffer manager.** The goal of the buffer manager is to ensure that as many data requests made by programs as possible are satisfied from data copied from secondary storage devices into the buffers. In effect, a program performs an input or an output operation using get or put statements; the buffer manager will be called on to respond to these input or output requests. It will check to see if the request for the data can be satisfied by reading from or writing to the existing buffers. If so, the input or output operation occurs between the program work area and buffers. If an input request cannot be satisfied, 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 makes 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 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 prefetching the next block of data and by batching write operations into the commit phase of a transaction.

We have assumed so far that the buffer manager uses buffers in physical memory. However, in a computer system that 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 11.16. Since the physical memory is managed by the memory management component of the operating system, a virtual buffer input by the buffer manager may

Figure 11.16 DBMS buffers in virtual memory.




SRINIVAS COLLEGE OF P.G. MANAGEMENT STUDIES
MANGALORE
575 001.

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 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. 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 to the buffer pages in virtual memory. The commit transaction operation can be considered 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 second phase is when the data buffers are written. 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 because the log is always forced during the first phase of the commit. With this scheme the undo log is not required, since no uncommitted modifications are reflected in the database.

In sequential processing of the database, the buffer manager prefetches the database pages. However, pages of data once used need not follow the locality property. A page once accessed is less likely to be accessed again. Hence, the buffer

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, the costs of 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 is delayed till the end of the transaction processing when all pages are written), then the costs of undo and redo are very low. Again each end of a transaction represents a checkpoint.

With an indirect update scheme where the end of the 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 it, the changes made by the transaction need not be rolled back. The rollback operation in this case consists of not propagating the modifications made by the transaction to the database. The same procedure will apply if the system aborts the transaction.

11.8 Disaster Recovery

Disaster refers to circumstances that 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 are the data stored in stable storage. The data stored in stable storage consist of the archival copy of the database and the archival log of the transactions on the database represented in the archival copy.

The **disaster 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, 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 result in performing archiving 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 reluctance to archive and the heavy cost of infrequent archiving 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 online access of the database. This updating can be performed on a 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 for a database system to work 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 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: failure without loss of data; failure with loss of volatile storage; failure with loss of nonvolatile storage; and failure with a loss of stable storage

The basic technique to implement 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, 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 time stored to stable storage. It contains the entire database in a quiescent mode and is made by simple dump routines to dump the physical database to stable storage. The purpose of the archival database is to recover from failures that involve loss of nonvolatile storage. 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, written in chronological order. 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 and indirect update with careful replacement: the shadow page scheme and the update via log scheme are two versions of the latter.

In the update-in-place scheme, the transaction updates the physical database and the modified record replaces the old record in the database. The write-ahead log strategy is used. The log information about the transaction modifications is written before update operations initiated by the transactions are 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; the transaction addresses the database using another table called the current 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 not interruptable by a system crash.

In the update via log scheme, the transaction is not allowed to modify the database. All changes to the database are deferred until the transaction commits. As in the update-in-place scheme, all modifications made by the transaction are logged. Since the database is not modified directly by the transaction, the old values do not have 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 every transaction in the archival log, the archival database being used as the initial version of the current database. The order of performing redo operations must be the same as the original order, hence the archival log file must be chronologically ordered.

Key Terms

reliable
failure
error
fault
fault-tolerant system
reliability
mean time between failures (MTBF)
mean time to repair (MTTR)
system availability
design error
poor quality control
overutilization
overloading
wearout
volatile storage
nonvolatile storage
system crash
permanent or stable storage
read/write head crash
archive database
user error
consistency error
system error
validity
deadlock
audit trail
journal
forward error recovery
backward error recovery
buffer
atomic operation
successful termination
suicidal termination
murderous termination
atomicity
consistency
isolation
durability
log
write-ahead log strategy
checkpoint
transaction-consistent checkpoint
action-consistent checkpoint
transaction-oriented checkpoint
undo
redo
quiescent
current database
materialized database
do
idempotent
transaction undo
transaction redo
global undo
global redo
update in place
indirect update
shadow page scheme
update via log
indirect page allocation
page table
shadow page table
current page table
buffer management
buffer manager
virtual memory
memory manager