least recently used (LRU)
two-phase commit
steal
not steal
force
no force
record-level logging
page-level logging
query language logging
disaster recovery

Exercises

11.1 What if anything can be done to recover the modifications made by partially completed transactions that are running at the time of a system crash? Can online transactions be 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 made to a recovery scheme if the transactions are nested? (In a nested transaction one transaction is contained within another transaction.)

11.4 In 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 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 that 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, a system crash will lead to the loss of a large number of transactions and a long recovery operation; if the checkpoint frequency is too high, the cost of checkpointing is very high. Can you suggest a method of reducing the frequency of checkpointing 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 online systems such as banking or airline reservations? Suggest a method to be used in such systems to restart active transactions after a system crash.

11.8 For a logging scheme based on a 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.

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

11.11 Define the following terms:

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

11.12 From the point of view of recovery, compare the shadow page scheme with the update in place with forced and no steal buffering.

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 checkpointing 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 Section 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 work in recovery was reported in (Oppe 68), (Chan 72), (Bjor 73), and (Davi 73). Analytical models for recovery and rollback and discussions are presented in (Chan 75). The concept of transaction and its management is presented in (Gray 78). The recovery system for System R is presented in (Gray 81a); the shadow page scheme used in System R is described in an earlier paper (Lori 77). (Verh 78) is an early survey article on database recovery; (Haer 83) and (Kohl 81) are more recent survey articles based on the transaction paradigm. An efficient logging scheme for the undo operation is discussed in (Reut 80). (Teng 84) discusses the buffer management function to optimize database performance for the DB2 relational database system.

The concept of nested transaction was discussed by (Gray 81a); more recent discussions are presented in (Moss 85).

Textbooks discussing the recovery operation are (Bern 88), (Date 83), (Date 86), and (Kort 86). Reliability concepts are presented in (Wied 83).

Bibliography

(Bern 88) P. Bernstein, V. Hadzilacos, & N. Goodman, *Concurrency Control and Recovery in Database Systems*. Reading, MA: Addison-Wesley, 1988.

(Bjor 73) L. A. Bjork, "Recovery Scenario for a DB/DC System," Proc. of the ACM Annual Conference, 1973, pp. 142–146.

(Chan 72) K. M. Chandy, & C. V. Ramamoorthy, "Rollback and Recovery Strategies for Computer Programs," *IEEE* . C-21(6), June 1972, pp. 546–555.

(Chan 75) K. M. Chandy, J. C. Browne, C. W. Dissly, & W. R. Uhrig, "Analytic Models for Rollback and Recovery Strategies in Data Base Systems," *IEEE* SE-1(1), March 1975, pp. 100–110.

(Date 83) C. J. Date, *An Introduction to Database Systems*, vol. 2, Reading, MA: Addison-Wesley, 1983.

(Date 86) C. J. Date, *An Introduction to Database Systems*, vol. 1,. 4th ed. Reading, MA: Addison-Wesley, 1986.

(Davi 73) J. C. Davies Jr., "Recovery Semantics for a DB/DC System," Proc. of the ACM Annual Conference, 1973, pp. 136–141.

(Gior 76) N. J. Giordano, & M. S. Schwartz. "Database Recovery at CMIC," Proc. ACM SIGMOD Conf. on Management of Data, June 1976, pp. 33–42.

(Gray 78) J. N. Gray, "Notes on Database Operating Systems," in R. Bayer et al., ed., Operating Systems: An Advanced Course. Berlin: Springer-Verlag, 1978.

(Gray 81) J. N. Gray, "The Transaction Concept: Virtues and Limitations," Proc. of the Intnl. Conf. on VLDB, 1981, pp. 144–154.

(Gray 81b) J. N. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, & I. Traiger, "The Recovery Manager of the System R Database Manager," *ACM Computing Surveys* 13(2), June 1981, pp. 223–242.

(Haer 83) T. Haerder, & A. Reuter, "Principles of Transaction(ru0,1n)Oriented Database Recovery," *ACM Computing Surveys* 15(4), December 1983, pp. 287–317.

(Kohl 81) K. H. Kohler, "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems," *ACM Computing Surveys* 13(2), June 1981, pp. 148–183.

(Kort 86) H. F. Korth, & A. Silberschatz, *Database System Concepts*. New York: McGraw-Hill, 1986.

(Lori 77) R. Lorie, "Physical Integrity in a Large Segmented Database," *ACM TODS* 2(1), March 1977, pp. 91–104.

(Lync 83) N. A. Lynch, "Multilevel Atomicity—A New Correctness Criterion for Database Concurrency Control," *ACM TODS* 8(4), December 1983, pp. 484–502.

(Moss 85) J. Moss, J. & B. Eliot, *Nested Transactions: An Approach to Reliable Distributed Computing*. Cambridge, MA: MIT Press, 1985.

(Oppe 68) G. Oppenheimer, K. P. Clancy, *Considerations of Software Protection and Recovery from Hardware Failures*. Washington, D.C.: FJCC, 1968.

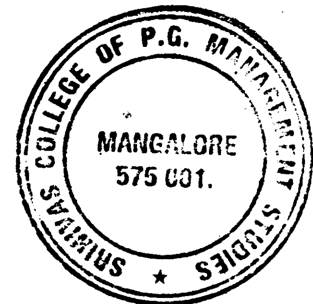
(Reut 80) A. Reuter, "A Fast Transaction-Oriented Logging Scheme For UNDO Recovery," *IEEE* SE 6(4), July 1980, pp. 348–356.

(Seve 76) D. G. Severance, & G. M. Lohman, "Differential Files: Their Application to the Maintenance of Large Databases," *ACM TODS* 1(3), September 1976, pp. 256–267.

(Teng 84) J. Z. Teng, & R. A. Gumaer, "Managing IBM Database 2 Buffers to Maximize Performance," *IBM Systems Journal* 23(2), 1984, pp. 211–218.

(Verh 78) J. S. M. Verhofstad, "Recovery Techniques for Database Systems," *ACM Computing Surveys* 10(2), June 1978, pp. 167–195.

(Wied 83) Gio Wiederhold, *Database Design*, 2nd ed. New York: McGraw-Hill, 1983.



the user. One method of enforcing mutual exclusion is by some type of locking mechanism that locks a shared resource (for example a data-item) used by a transaction for the duration of its usage by the transaction. The locked data-item can only be used by the transaction that locked it. The other concurrent transactions are locked out and have to wait their turn at using the data-item. However, a locking scheme must be fair. This requires that the lock manager, which is the DBMS subsystem managing the locks, must not cause some concurrent transaction to be permanently blocked from using the shared resource. This is referred to as avoiding the **starvation** or livelock situation. The other danger to be avoided is that of deadlock, wherein a number of transactions are waiting in a circular chain, each waiting for the release of resources held by the next transaction in the chain.

In other methods of concurrency control, some form of a priori ordering with a single or many versions of data is used. These methods are called timestamp ordering and multiversion schemes. The optimistic approach, on the other hand, assumes that the data-items used by concurrent transactions are most likely be disjoint.

Concurrency and Possible Problems

In the last chapter we stressed that a correct transaction, when completed, leaves the database in a consistent state provided that the database was in a consistent state at the start of the transaction. Nevertheless, during the life of a transaction, the database could be inconsistent, although if the inconsistencies are not accessible to other transactions, they would not cause a problem.

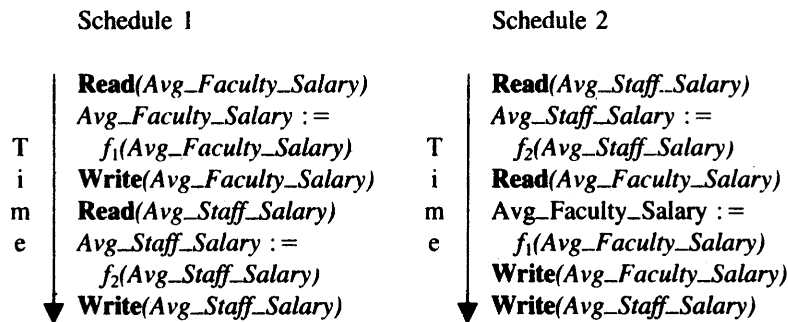
In the case of concurrent operations, where a number of transactions are running and using the database, we cannot make any assumptions about the order in which the statements belonging to different transactions will be executed. The order in which these statements are executed is called a **schedule.** Consider the two transactions in Figure 12.1. Each transaction reads some data-item, performs some operations on the data-item that could change its value, and then writes out the modified data-item.

In Figure 12.1 and in subsequent examples in this chapter, we assume that the read operation reads in the database value of the named variable to a local variable with an identical name. Any modifications by a transaction are made on this local copy. The modifications made by the transactions are indicated by the operators f_1 and f_2 in Figure 12.1. These modifications are not reflected in the database until the write operation is executed, at which point the modifications in the value of the

Figure 12.1 Two concurrent transactions.

Transaction T_1	Transaction T_2
Read(*Avg_Faculty_Salary*)	**Read**(*Avg_Staff_Salary*)
Avg_Faculty_Salary :=	*Avg_Staff_Salary* :=
f_1(*Avg_Faculty_Salary*)	f_2(*Avg_Staff_Salary*)
Write(*Avg_Faculty_Salary*)	**Write**(*Avg_Staff_Salary*)

Figure 12.2 Possible interleaving of concurrent transactions of Figure 12.1.



named variable are said to be committed. In effect the write operation is a signal for committing the modifications and reflecting the changes to the physical database.

Figure 12.2 gives two possible schedules for executing the transactions of Figure 12.1 in an interleaved manner. Since the transactions of Figure 12.1 are accessing and modifying distinct data-items, *(Avg_Faculty_Salary, Avg_Staff_Salary)*, there is no problem in executing these transactions concurrently. In other words, regardless of the order of interleaving of the statements of these transactions, we will get a consistent database on the termination of these transactions.

12.1.1 Lost Update Problem

Consider the transactions of Figure 12.3. These transactions are accessing the same data-item A. Each of the transactions modifies the data-item and writes it back. Again let us consider a number of possible interleavings of the execution of the statements of these transactions. These schedules are given in Figure 12.4.

Starting with 200 as the initial value of A, let us see what the value of A would be if the transactions are run without any interleaving. In other words, the transactions are run to completion, without any interruptions, one at a time in a serial manner. If transaction T_3 is run first, then at the end of the transaction the value of A will have changed from 200 to 210. Running transaction T_4 after the completion of T_3 will change the value of A from 210 to 231. Running the transactions in the

Figure 12.3 Two transactions modifying the same data-item.

Figure 12.4 Two schedules for transactions of Figure 12.3.

Schedule 1	Transaction T_3	Transaction T_4	Value of A
Read(A)		**Read**(A)	200
$A := A * 1.1$		$A := A * 1.1$	
Read(A)	**Read**(A)		
$A := A + 10$	$A := A + 10$		
Write(A)	**Write**(A)		210
Write(A)		**Write**(A)	220

(a)

Schedule 2	Transaction T_3	Transaction T_4	Value of A
Read(A)	**Read**(A)		200
$A := A + 10$	$A := A + 10$		
Read(A)		**Read**(A)	
$A := A * 1.1$		$A := A * 1.1$	
Write(A)		**Write**(A)	220
Write(A)	**Write**(A)		210

(b)

(Time runs downward in both schedules.)

order T_4 followed by T_3 result in a final value for A of 230. The result obtained with neither of the two interleaved execution schedules of Figure 12.4 agrees with either of the results of executing these same transactions serially. Obviously something is wrong!

In each of the schedules given in Figure 12.4, we have lost the update made by one of the transactions. In schedule 1, the update made by transaction T_3 is lost; in schedule 2, the update made by transaction T_4 is lost. Each schedule exhibits an example of the so-called **lost update** problem of the concurrent execution of a number of transactions.

It is obvious that the reason for the lost update problem is that even though we have been able to enforce that the changes made by one concurrent transaction are not accessible by the other transactions until it commits, we have not enforced the atomicity requirement. This demands that only one transaction can modify a given data-item at a given time and other transactions should be locked out from even viewing the unmodified value (in the database) until the modifications (being made to a local copy of the data) are committed to the database.

12.1.2 Inconsistent Read Problem

The lost update problem was caused by concurrent modifications of the same data-item. However, concurrency can also cause problems when only one transaction modifies a given set of data while that set of data is being used by other transactions.