Chapter 14. Backup and Recovery

Table of contents

Objectives

At the end of this chapter you should be able to:

- Describe a range of causes of database failure, and explain mechanisms available to deal with these.
- Understand a range of options available for the design of database backup procedures.

- Analyse the problems of data management in a concurrent environment.
- Be able to discuss the software mechanisms you would expect to be provided for recovery in a large, multi-user database environment.

Relationship to other chapters

The design of suitable backup and recovery procedures will usually be carried out by the database administrator (DBA) or DBA group, probably in conjunction with representatives of management and application developers. The way in which the databases in use provide for the concurrent processing of transactions, covered in the chapter Concurrency Control, will have an impact on the design of the backup and recovery procedures required.

Context

In the previous chapter, Concurrency Control, we discussed the different causes of failure such as transaction errors and system crashes. In this chapter we will introduce some of the techniques that can be used to recover from transaction failures.

We will first introduce some concepts that are used by recovery processes such as the system log, checkpoints and commit points. We then outline the recovery procedures. The process of rolling back (undoing) the effect of a transaction will be discussed in detail.

We will present recovery techniques based on deferred update, also known as the NO-UNDO/REDO technique, and immediate update, which is known as UNDO/REDO. We also discuss the technique known as shadowing or shadow paging, which can be categorised as a NO-UNDO/NO-REDO algorithm. Recovery in multidatabase transactions is briefly discussed in the chapter. Techniques for recovery from catastrophic failure are also discussed briefly.

Introduction

In parallel with this chapter, you should read Chapter 22 of Ramez Elmasri and Shamkant B. Navathe, " FUNDAMENTALS OF Database Systems", (7th edn.).

Database systems, like any other computer system, are subject to failures. Despite this, any organisation that depends upon a database must have that database available when it is required. Therefore, any DBMS intended for a serious business or organisational user must have adequate facilities for fast recovery after failure. In particular, whenever a transaction is submitted to a DBMS for execution, the system must ensure that either all the operations in the

transaction are completed successfully and their effect is recorded permanently in the database, or the transaction has no effect whatsoever on the database or on any other transactions.

The understanding of the methods available for recovering from such failures are therefore essential to any serious study of database. This chapter describes the methods available for recovering from a range of problems that can occur throughout the life of a database system. These mechanisms include automatic protection mechanisms built into the database software itself, and non-automatic actions available to people responsible for the running of the database system, both for the backing up of data and recovery for a variety of failure situations.

Recovery techniques are intertwined with the concurrency control mechanisms: certain recovery techniques are best used with specific concurrency control methods. Assume for the most part that we are dealing with large multi-user databases. Small systems typically provide little or no support for recovery; in these systems, recovery is regarded as a user problem.

A typical recovery problem

Data updates made by a DBMS are not automatically written to disk at each synchronisation point. Therefore there may be some delay between the commit and the actual disk writing (i.e. regarding the changes as permanent and worthy of being made to disk). If there is a system failure during this delay, the system must still be able to ensure that these updates reach the disk copy of the database. Conversely, data changes that may ultimately prove to be incorrect, made for example by a transaction that is later rolled back, can sometimes be written to disk. Ensuring that only the results of complete transactions are committed to disk is an important task, which if inadequately controlled by the DBMS may lead to problems, such as the generation of an inconsistent database. This particular problem can be clearly seen in the following example.

Suppose we want to enter a transaction into a customer order. The following actions must be taken:

START

1. *Change the customer record with the new order data*
2. *Change the salesperson record with the new order data*
3. *Insert a new order record into the database*

STOP

And the initial values are shown in the figure below:

CUSTOMER:

C-No	Order-No	Description	Cost
123	1,000	400 Tennis balls	£2,400

SALESPERSON:

Name	Total-Sales
Jones	£3,200

ORDERS:

Order-No
1000
2000
3000
4000
5000

If only operations 1 and 2 are successfully performed, this results in the values shown in the figure below:

CUSTOMER:

C-No	Order-No	Description	Cost
123	1,000	400 Tennis balls	£2,400
123	8,000	250 Cricket balls	£6,500

SALESPERSON:

Name	Total-Sales
Jones	£9,700

ORDERS:

Order-No
1000
2000
3000
4000
5000

This database state is clearly unacceptable as it does not accurately reflect reality; for example, a customer may receive an invoice for items never sent, or a salesman may make commission on items never received. It is better to treat the whole procedure (i.e. from START to STOP) as a complete transaction and not commit any changes to the database until STOP has been successfully reached.

Transaction logging

System log

The recovery manager overcomes many of the potential problems of transaction failure by a variety of techniques. Many of these are heavily dependent upon the existence of a special file known as a system log, or simply a log (also sometimes called a journal or audit trail). It contains information about the start and end of each transaction and any updates which occur in the transaction. The log keeps track of all transaction operations that affect the values of database items. This information may be needed to recover from transaction failure. The log is kept on disk (apart from the most recent log block that is in the process of being generated, this is stored in the main memory buffers). Thus, the majority of the log is not affected by failures, except for a disk failure or catastrophic failure. In addition, the log is periodically backed up to archival storage (e.g. tape) to guard against such catastrophic failures. The types of entries that are written to the log are described below. In these entries, T refers to a unique transaction identifier that is generated automatically by the system and used to uniquely label each transaction.

- **start_transaction(T):** This log entry records that transaction T starts the execution.
- **read_item(T, X):** This log entry records that transaction T reads the value of database item X.
- **write_item(T, X, old_value, new_value):** This log entry records that transaction T changes the value of the database item X from old_value to new_value. The old value is sometimes known as a before image of X, and the new value is known as an after image of X.
- **commit(T):** This log entry records that transaction T has completed all accesses to the database successfully and its effect can be committed (recorded permanently) to the database.
- **abort(T):** This records that transaction T has been aborted.
- **checkpoint:** This is an additional entry to the log. The purpose of this entry will be described in a later section.

Some protocols do not require that read operations be written to the system log, in which case, the overhead of recording operations in the log is reduced, since fewer operations – only write – are recorded in the log. In addition, some protocols require simpler write entries that do not include new_value.

Because the log contains a record of every write operation that changes the value of some database item, it is possible to undo the effect of these write operations of a transaction T by tracing backward through the log and resetting all items changed by a write operation of T to their old_values. We can also redo the

effect of the write operations of a transaction T by tracing forward through the log and setting all items changed by a write operation of T to their new_values. Redoing the operations of a transaction may be required if all its updates are recorded in the log but a failure occurs before we can be sure that all the new_values have been written permanently in the actual database.

Committing transactions and force-writing

A transaction T reaches its commit point when all its operations that access the database have been executed successfully; that is, the transaction has reached the point at which it will not abort (terminate without completing). Beyond the commit point, the transaction is said to be committed, and its effect is assumed to be permanently recorded in the database. Commitment always involves writing a commit entry to the log and writing the log to disk. At the time of a system crash, we search back in the log for all transactions T that have written a start_transaction(T) entry into the log but have not written a commit(T) entry yet; these transactions may have to be rolled back to undo their effect on the database during the recovery process. Transactions that have written their commit(T) entry in the log must also have recorded all their write operations in the log - otherwise they would not be committed - so their effect on the database can be redone from the log entries.

Notice that the log file must be kept on disk. At the time of a system crash, only the log entries that have been written back to disk are considered in the recovery process, because the contents of main memory may be lost. Hence, before a transaction reaches its commit point, any portion of the log that has not been written to the disk yet must now be written to the disk. This process is called force-writing of the log file, before committing a transaction. A commit does not necessarily involve writing the data items to disk; this depends on the recovery mechanism in use.

A commit is not necessarily required to initiate writing of the log file to disk. The log may sometimes be written back automatically when the log buffer is full. This happens irregularly, as usually one block of the log file is kept in main memory until it is filled with log entries and then written back to disk, rather than writing it to disk every time a log entry is added. This saves the overhead of multiple disk writes of the same information.

Checkpoints

In the event of failure, most recovery managers initiate procedures that involve redoing or undoing operations contained within the log. Clearly, not all operations need to be redone or undone, as many transactions recorded on the log will have been successfully completed and the changes written permanently to disk. The problem for the recovery manager is to determine which operations

need to be considered and which can safely be ignored. This problem is usually overcome by writing another kind of entry in the log: the checkpoint entry.

The checkpoint is written into the log periodically and always involves the writing out to the database on disk the effect of all write operations of committed transactions. Hence, all transactions that have their commit(T) entries in the log before a checkpoint entry will not require their write operations to be redone in case of a system crash. The recovery manager of a DBMS must decide at what intervals to take a checkpoint; the intervals are usually decided on the basis of the time elapsed, or the number of committed transactions since the last checkpoint. Performing a checkpoint consists of the following operations:

- Suspending executions of transactions temporarily;
- Writing (force-writing) all modified database buffers of committed transactions out to disk;
- Writing a checkpoint record to the log; and
- Writing (force-writing) all log records in main memory out to disk.

A checkpoint record usually contains additional information, including a list of transactions active at the time of the checkpoint. Many recovery methods (including the deferred and immediate update methods) need this information when a transaction is rolled back, as all transactions active at the time of the checkpoint and any subsequent ones may need to be redone.

In addition to the log, further security of data is provided by generating backup copies of the database, held in a separate location to guard against destruction in the event of fire, flood, disk crash, etc.

Undoing

If a transaction crash does occur, then the recovery manager may undo transactions (that is, reverse the operations of a transaction on the database). This involves examining a transaction for the log entry write_item(T, x, old_value, new_value) and setting the value of item x in the database to old-value. Undoing a number of write_item operations from one or more transactions from the log must proceed in the reverse order from the order in which the operations were written in the log.

Redoing

Redoing transactions is achieved by examining a transaction's log entry and for every write_item(T, x, old_value, new_value) entry, the value of item x in the database is set to new_value. Redoing a number of transactions from the log must proceed in the same order in which the operations were written in the log.

The redo operation is required to be idempotent; that is, executing it over and over is equivalent to executing it just once. In fact, the whole recovery process should be idempotent. This is so because, if the system were to fail during the recovery process, the next recovery attempt might redo certain write_item operations that had already been redone during the previous recovery process. The result of recovery from a crash during recovery should be the same as the result of recovering when there is no crash during recovery. Of course, repeating operations, as long as they are done in the correct way, should never leave the database in an inconsistent state, although as we have seen the repetitions may be unnecessary.

It is only necessary to redo the last update of x from the log during recovery, because the other updates would be overwritten by this last redo. The redo algorithm can be made more efficient by starting from the end of the log and working backwards towards the last checkpoint. Whenever an item is redone, it is added to a list of redone items. Before redo is applied to an item, the list is checked; if the item appears on the list, it is not redone, since its last value has already been recovered.

Activity 1 - Looking up glossary entries

In the Transaction Logging section of this chapter, the following terms have glossary entries:

- system log
- commit point
- checkpoint
- force-writing

1. In your own words, write a short definition for each of these terms.
2. Look up and make notes of the definition of each term in the module glossary.
3. Identify (and correct) any important conceptual differences between your definition and the glossary entry.

Review question 1

1. Unfortunately, transactions fail frequently, and they do so due to a variety of causes. Review the chapter on Concurrency Control, and discuss the different causes of the transaction failures.
2. What is meant by a system log? Discuss how a system log is needed in the recovery process.
3. Discuss the actions involved in writing a checkpoint entry.

4. Discuss how undo and redo operations are used in the recovery process.

Recovery outline

Recovery from transaction failures usually means that the database is restored to some state from the past, so that a correct state – close to the time of failure – can be reconstructed from that past state. To do this, the system must keep information about changes to data items during transaction execution outside the database. This information is typically kept in the system log. It is important to note that a transaction may fail at any point, e.g. when data is being written to a buffer or when a log is being written to disk. All recovery mechanisms must be able to cope with the unpredictable nature of transaction failure. Significantly, the recovery phase itself may fail; therefore, the recovery mechanism must also be capable of recovering from failure during recovery.

A typical strategy for recovery may be summarised based on the type of failures.

Recovery from catastrophic failures

The main technique used to handle catastrophic failures including disk crash is that of database backup. The whole database and the log are periodically copied onto a cheap storage medium such as magnetic tapes. In case of a catastrophic system failure, the latest backup copy can be reloaded from the tape to the disk, and the system can be restarted.

To avoid losing all the effects of transactions that have been executed since the last backup, it is customary to back up the system log by periodically copying it to magnetic tape. The system log is usually substantially smaller than the database itself and hence can be backed up more frequently. When the system log is backed up, users do not lose all transactions they have performed since the last database backup. All committed transactions recorded in the portion of the system log that has been backed up can have their effect on the database reconstructed. A new system log is started after each database backup operation. Hence, to recover from disk failure, the database is first recreated on disk from its latest backup copy on tape. Following that, the effects of all the committed transactions whose operations have been entered in the backed-up copy of the system log are reconstructed.

Recovery from non-catastrophic failures

When the database is not physically damaged but has become inconsistent due to non-catastrophic failure, the strategy is to reverse the changes that caused the inconsistency by undoing some operations. It may also be necessary to redo some operations that could have been lost during the recovery process, or for

some other reason, in order to restore a consistent state of the database. In this case, a complete archival copy of the database is not required; rather, it is sufficient that the entries kept in the system log are consulted during the recovery.

There are two major techniques for recovery from non-catastrophic transaction failures: deferred updates and immediate updates. The deferred update techniques do not actually update the database until after a transaction reaches its commit point; then the updates are recorded in the database. Before commit, all transaction updates are recorded in the local transaction workspace. During commit, the updates are first recorded persistently in the log and then written to the database. If a transaction fails before reaching its commit point, it will not have changed the database in any way, so UNDO is not needed. It may be necessary to REDO the effect of the operations of a committed transaction from the log, because their effect may not yet have been written in the database. Hence, deferred update is also known as the NO-UNDO/REDO algorithm.

In the immediate update techniques, the database may be updated by some operations of a transaction before the transaction reaches its commit point. However, these operations are typically recorded in the log on disk by force-writing before they are applied to the database, making recovery still possible. If a transaction fails after recording some changes in the database but before reaching its commit point, the effect of its operations on the database must be undone; that is, the transaction must be rolled back. In the general case of immediate update, both undo and redo are required during recovery, so it is known as the UNDO/REDO algorithm.

Transaction rollback

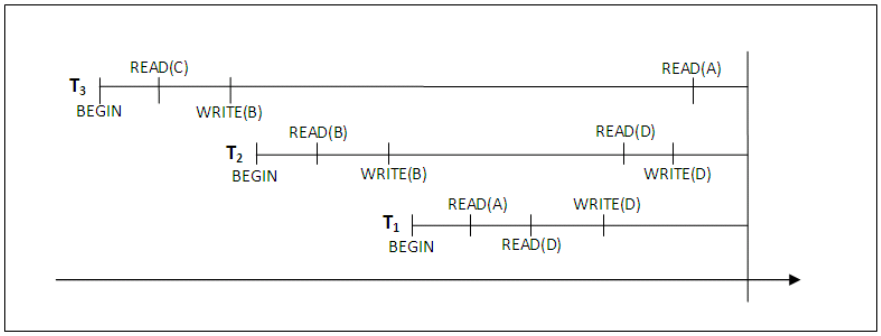
If a transaction fails for whatever reason after updating the database, it may be necessary to roll back or UNDO the transaction. Any data item values that have been changed by the transaction must be returned to their previous values. The log entries are used to recover the old values of data items that must be rolled back.

If a transaction T is rolled back, any transaction S that has, in the interim, read the value of some data item X written by T must also be rolled back. Similarly, once S is rolled back, any transaction R that has read the value of some item Y written by S must also be rolled back; and so on. This phenomenon is called cascading rollback. Cascading rollback, understandably, can be quite time-consuming. That is why most recovery mechanisms are designed such that cascading rollback is never required.

The table below shows an example where cascading rollback is required. The read and write operations of three individual transactions are shown below:

T_1	T_2	T_3
Read_item(A);	read_item(B);	Read_item(C);
Read_item(D);	write_item(B);	Write_item(B);
Write_item(D);	read_item(D);	Read_item(A);
	write_item(D);	Write_item(A);

The diagram below graphically shows the operations of different transactions along the time axis:



The figure below shows the system log at the point of a system crash for a particular execution schedule of these transactions. The values of A, B, C and D, which are used by the transactions, are shown to the right of the system log entries. At the point of system crash, transaction T3 has not reached its conclusion and must be rolled back. The write operations of T3, marked by a single *, are the operations that are undone during transaction rollback.

Log	A	B	C	D
	30	15	40	20
start_transaction(T_3)				
read_item(T_3, C)				
*write_item(T_3, B, 15, 12)		12		
start_transaction(T_2)				
read_item(T_2, B)				
**write_item(T_2, B, 12, 18)		18		
start_transaction(T_1)				
read_item(T_1, A)				
read_item(T_1, D)				
write_item(T_1, D, 20, 25)				25
read_item(T_2, D)				
**write_item(T_2, D, 25, 26)				26
read_item(T_3, A)				
System crash				

* T_3 is rolled back because it did not reach its commit point when system crash happens.

** T_2 is rolled back because it reads the value of item B written by T_3.

The rest of the write entries in the log are redone.

We must now check for cascading rollback. In the diagram above, which shows

transactions along the time axis, we see that transaction T2 reads the value B which was written by T3; this can also be determined by examining the log. Because T3 is rolled back, T2 must also be rolled back. The write operations of T2, marked by ** in the log, are the ones that are undone. Note that only write operations need to be undone during transaction rollback; read operations are recorded in the log only to determine whether cascading rollback of additional transactions is necessary. If rollback of transactions is never required by the recovery method, we can keep more limited information in the system log. There is also no need to record any read_item operations in the log, because these are needed only for determining cascading rollback.

Review question 2

1. What is catastrophic failure? Discuss how databases can recover from catastrophic failures.

2. What is meant by transaction rollback? Why is it necessary to check for cascading rollback?

3. Compare deferred update with immediate update techniques by filling in the following blanks.

   The deferred update techniques do not actually update the database until a transaction reaches its commit point; then the updates are recorded in the database. Before commit, all transaction updates are recorded in the local transaction workspace. During commit, the updates are first recorded persistently in the _____ and then written to the _____. If a transaction fails before reaching its commit point, it will not have changed the database in any way, so is not needed. It may be necessary to _____ the effect of the operations of a committed transaction in the log, because their effect may not yet have been written in the database. Hence, deferred update is also known as the _____ algorithm. In the immediate update techniques, the database may be updated by some operations of a transaction _____ the transaction reaches its commit point. However, these operations are typically recorded in the log on disk before they are applied to the database, making recovery still possible. If a transaction fails after recording some changes in the database but before reaching its commit point, the effect of its operations on the database must be _____; that is, the transaction must be _____. In the general case of immediate update, both _____ and _____ are required during recovery, so it is known as the _____ algorithm.

Recovery techniques based on deferred update

Deferred update

The idea behind deferred update is to defer or postpone any actual updates to the database itself until the transaction completes its execution successfully and reaches its commit point. During transaction execution, the updates are recorded only in the log and in the transaction workspace. After the transaction reaches its commit point and the log is force-written to disk, the updates are recorded in the database itself. If a transaction fails before reaching its commit point, there is no need to undo any operations, because the transaction has not affected the database in any way.

The steps involved in the deferred update protocol are as follows:

1. When a transaction starts, write an entry start_transaction(T) to the log.
2. When any operation is performed that will change values in the database, write a log entry write_item(T, x, old_value, new_value).
3. When a transaction is about to commit, write a log record of the form commit(T); write all log records to disk.
4. Commit the transaction, using the log to write the updates to the database; the writing of data to disk need not occur immediately.
5. If the transaction aborts, ignore the log records and do not write the changes to disk.

The database is never updated until after the transaction commits, and there is never a need to UNDO any operations. Hence this technique is known as the NO-UNDO/REDO algorithm. The REDO is needed in case the system fails after the transaction commits but before all its changes are recorded in the database. In this case, the transaction operations are redone from the log entries. The protocol and how different entries are affected can be best summarised as shown:

Log entry	Log written to disk	Changes written to database buffer	Changes written on disk
start_transaction(T)	No	N/A	N/A
read_item(T, x)	No	N/A	N/A
write_item(T, x)	No	No	No
commit(T)	Yes	Yes	*Yes
checkpoint	Yes	Undefined	Yes(of committed Ts)

*Yes: writing back to disk may occur not immediately.

Deferred update in a single-user environment

We first discuss recovery based on deferred update in single-user systems, where no concurrency control is needed, so that we can understand the recovery process independently of any concurrency control method. In such an environment, the recovery algorithm can be rather simple. It works as follows.

Use two lists to maintain the transactions: the committed transactions list, which contains all the committed transactions since the last checkpoint, and the active transactions list (at most one transaction falls in this category, because the system is a single-user one). Apply the REDO operation to all the write_item operations of the committed transactions from the log in the order in which they were written to the log. Restart the active transactions.

The REDO procedure is defined as follows:

Redoing a write_item operation consists of examining its log entry write_item(T, x, old_value, new_value) and setting the value of item x in the database to new_value. The REDO operation is required to be idempotent, as discussed before.

Notice that the transaction in the active list will have no effect on the database because of the deferred update protocol, and is ignored completely by the recovery process. It is implicitly rolled back, because none of its operations were reflected in the database. However, the transaction must now be restarted, either automatically by the recovery process or manually by the user.

The method's main benefit is that any transaction operation need never be undone, as a transaction does not record its changes in the database until it reaches its commit point.

The protocol is summarised in the diagram below:

Action	Entry in log	
	start_transaction(T)	commit(T)
Re-submit	Yes	No
Redo	Yes	Yes

The diagram below shows an example of recovery in a single-user environment, where the first failure occurs during execution of transaction T2. The recovery process will redo the write_item(T1, D, 20) entry in the log by resetting the value of item D to 20 (its new value). The write(T2, …) entries in the log are ignored by the recovery process because T2 is not committed. If a second failure occurs during recovery from the first failure, the same recovery process is repeated from start to finish, with identical results.

start_transaction(T1)	
write_item(T1, D, 20)	
commit(T1)	
start_transaction(T2)	
write_item(T2, B, 10)	
write_item(T2, D 25)	← System crash

T_1	T_2
read_item(A)	read_item(B)
read_item(D)	write_item(B)
write_item(D)	read_item(D)
	write_item(D)

Deferred update in a multi-user environment

For a multi-user system with concurrency control, the recovery process may be more complex, depending on the protocols used for concurrency control. In many cases, the concurrency control and recovery processes are interrelated. In general, the greater the degree of concurrency we wish to achieve, the more difficult the task of recovery becomes.

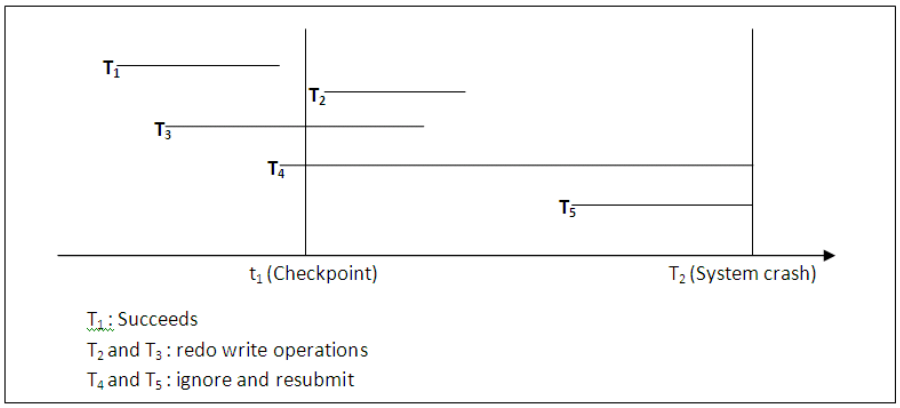
Consider a system in which concurrency control uses two-phase locking (basic 2PL) and prevents deadlock by pre-assigning all locks to items needed by a transaction before the transaction starts execution. To combine the deferred update methods for recovery with this concurrency control technique, we can keep all the locks on items in effect until the transaction reaches its commit point. After that, the locks can be released. This ensures strict and serialisable schedules. Assuming that checkpoint entries are included in the log, a possible

recovery algorithm for this case is given below.

Use two lists of transactions maintained by the system: the committed transactions list which contains all committed transactions since the last checkpoint, and the active transactions list. REDO all the write operations of the committed transactions from the log, in the order in which they were written into the log. The transactions in the active list that are active and did not commit are effectively cancelled and must be resubmitted.

The REDO procedure is the same as defined earlier in the deferred update in the single-user environment.

The diagram below shows an example schedule of executing transactions. When the checkpoint was taken at time t1, transaction T1 had committed, whereas transaction T3 and T4 had not. Before the system crash at time t2, T3 and T2 were committed but not T4 and T5. According to the deferred update method, there is no need to redo the write operations of transaction T1 or any transactions committed before the last checkpoint time t1. Write operations of T2 and T3 must be redone, however, because both transactions reached their commit points after the last checkpoint. Recall that the log is force-written before committing a transaction. Transaction T4 and T5 are ignored: they are effectively cancelled and rolled back because none of their write operations were recorded in the database under the deferred update protocol.



Transaction actions that do not affect the database

In general, a transaction will have actions that do not affect the database, such as generating and printing messages or reports from information retrieved from the database. If a transaction fails before completion, we may not want the user to get these reports, since the transaction has failed to complete. Hence, such

reports should be generated only after the transaction reaches its commit point. A common method of dealing with such actions is to issue the commands that generate the reports but keep them as batch jobs. The batch jobs are executed only after the transaction reaches its commit point. If the transaction does not reach its commit point because of a failure, the batch jobs are cancelled.

Exercise 1: Deferred update protocol

Given the operations of the four concurrent transactions in (1) below and the system log at the point of system crash in (2), discuss how each transaction recovers from the failure using deferred update technique.

1. The read and write operations of four transactions:

T_1	T_2	T_3	T_4
read_item(A)	read_item(B)	read_item(A)	read_item(B)
read_item(D)	write_item(B)	write_item(A)	write_item(B)
write_item(D)	read_item(D)	read_item(C)	read_item(A)
	write_item(D)	write_item(C)	write_item(A)

2. System log at the point of crash:

start_transaction(T_1)	
write_item(T_1, D, 20)	
commit(T_1)	
checkpoint	
start_transaction(T_4)	
write_item(T_4, B, 15)	
commit(T_4)	
start_transaction(T_2)	
write_item(T_2, B, 12)	
start_transaction(T_3)	
write_item(T_3, A, 30)	
write_item(T_2, D, 25)	← System crash

Exercise 2: Recovery management using deferred update with incremental log

Below, a schedule is given for five transactions A, B, C, D and E.

Assume the initial values for the variables are a=1, b=2, c=3, d=4 and e=5.

Using an incremental log with deferred updates, for each operation in each of the transactions, show:

1. The log entries.
2. Whether the log is written to disk.
3. Whether the output buffer is updated.
4. Whether the DBMS on disk is updated.
5. The values of the variables on the disk.

Discuss how each transaction recovers from the failure.

A	B	C	D	E	Log entries	Log to disc	Buffer changed	DBMS on disk changed	a	b	c	d	e
									1	2	3	4	5
start A	start A												
read value a	read value a												
a := a * 2	a := a * 2												
write value a	write value a												
commit A	commit A												
		start B											
		read value b											
		b := b * 2											
		write value b											
			start C										
			read value c										
			c := c * 2										
			write value c										
						Check point							
		b := 0											
		write value b											
		commit B											
				start D									
				read value d									
				d := d * 2									
				write value d									
				commit D									
					start E								
					read value e								
					e := e * 2								
					write value e								
						FAIL URE							

Review question 3

1. Use your own words to describe the deferred update method for recovery management in a multi-user environment. Complete the following table to show how the deferred update protocol affects the log on disk, database buffer and database on disk.

Log entry	Log written to disk	Changes written to database buffer	Changes written on disk
Start_transaction(T)			
Read_item(T, x)			
Write_item(T, x)			
Commit(T)			
Checkpoint			

2. How can recovery handle transaction operations that do not affect the database, such as the printing of reports by the transaction?

Recovery techniques based on immediate update

Immediate update

In the immediate update techniques, the database may be updated by the operations of a transaction immediately, before the transaction reaches its commit point. However, these operations are typically recorded in the log on disk by force-writing before they are applied to the database, so that recovery is possible.

When immediate update is allowed, provisions must be made for undoing the effect of update operations on the database, because a transaction can fail after it has applied some updates to the database itself. Hence recovery schemes based on immediate update must include the capability to roll back a transaction by undoing the effect of its write operations.

1. When a transaction starts, write an entry start_transaction(T) to the log;
2. When any operation is performed that will change values in the database, write a log entry write_item(T, x, old_value, new_value);
3. Write the log to disk;
4. Once the log record is written, write the update to the database buffers;
5. When convenient, write the database buffers to the disk;
6. When a transaction is about to commit, write a log record of the form commit(T);
7. Write the log to disk.

The protocol and how different entries are affected can be best summarised below:

Log entry	Log written to disk	Changes written to database buffer	Changes written on disk
start_transaction(T)	No	N/A	N/A
read_item(T, x)	No	N/A	N/A
write_item(T, x)	Yes	Yes	*Yes
commit(T)	Yes	Undefined	Undefined
Checkpoint	Yes	Undefined	Yes(of committed Ts)

*Yes: writing back to disk may not occur immediately

In general, we can distinguish two main categories of immediate update algorithms. If the recovery technique ensures that all updates of a transaction

are recorded in the database on disk before the transaction commits, there is never a need to redo any operations of committed transactions. Such an algorithm is called UNDO/NO-REDO. On the other hand, if the transaction is allowed to commit before all its changes are written to the database, we have the UNDO/REDO method, the most general recovery algorithm. This is also the most complex technique. Recovery activities are summarised below:

Action	Entry in log	
	start_transaction(T)	commit(T)
Undo and resubmit T	Yes	No
Redo	Yes	Yes

Immediate update in a single-user environment

We first consider a single-user system so that we can examine the recovery process separately from concurrency control. If a failure occurs in a single-user system, the executing transaction at the time of failure may have recorded some changes in the database. The effect of all such operations must be undone as part of the recovery process. Hence, the recovery algorithm needs an UNDO procedure, described subsequently, to undo the effect of certain write operations that have been applied to the database following examination of their system log entry. The recovery algorithm also uses the redo procedure defined earlier. Recovery takes place in the following way.

Use two lists of transaction maintained by the system: the committed transactions since the last checkpoint, and the active transactions (at most one transaction will fall in this category, because the system is single user). Undo all the write operations of the active transaction from the log, using the UNDO procedure described hereafter. Redo all the write operations of the committed transactions from the log, in the order in which they were written in the log, using the REDO procedure.

The UNDO procedure is defined as follows:

Undoing a write operation consists of examining its log entry write_item(T, x, old_value, new_value) and setting the value of x in the database to old_value. Undoing a number of such write operations from one or more transactions from the log must proceed in the reverse order from the order in which the operations were written in the log.

Immediate update in a multi-user environment

When concurrency execution is permitted, the recovery process again depends on the protocols used for concurrency control. The procedure below outlines a recovery technique for concurrent transactions with immediate update. Assume that the log includes checkpoints and that the concurrency control protocol produces strict schedules – as, for example, the strict 2PL protocol does. Recall that a strict schedule does not allow a transaction to read or write an item unless the transaction that last wrote the item has committed. However, deadlocks can occur in strict 2PL, thus requiring UNDO of transactions.

Use two lists of transaction maintained by the system: the committed transactions since the last checkpoint, and the active transactions. Undo all the write operations of the active (uncommitted) transaction from the log, using the UNDO procedure. The operations should be undone in the reverse of the order in which they were written into the log. Redo all the write operations of the committed transactions from the log, in the order in which they were written in the log, using the REDO procedure.

Exercise 3: Immediate update protocol

Given the same transactions and system log in exercise 1, discuss how each transaction recovers from the failure using immediate update technique.

Exercise 4: Recovery management using immediate update with incremental log

The same schedule and initial values of the variables in exercise 2 are given; use the immediate update protocol for recovery to show how each transaction recovers from the failure.

Review question 4

1. Use your own words to describe the immediate update method for recovery management in a multi-user environment. Complete the following table to show how the immediate update protocol affects the log on disk, database buffer and database on disk.

Log entry	Log written to disk	Changes written to database buffer	Changes written on disk
Start_transaction(T)			
Read_item(T, x)			
Write_item(T, x)			
Commit(T)			
Checkpoint			

2. In general, we can distinguish two main categories of immediate update algorithms. If the recovery technique ensures that all updates of a transac-

tion are recorded in the database on disk before the transaction commits, there is never a need to redo any operations of committed transactions. Such an algorithm is called UNDO/NO-REDO. On the other hand, if the transaction is allowed to commit before all its changes are written to the database, we have the UNDO/REDO method.

Recovery in multidatabase transactions

So far, we have implicitly assumed that a transaction accesses a single database. In some cases a single transaction, called a multidatabase transaction, may require access to multiple database. These databases may even be stored on different types of DBMSs; for example, some DBMSs may be Relational, whereas others are hierarchical or network DBMSs. In such a case, each DBMS involved in the multidatabase transaction will have its own recovery technique and transaction manager separate from those of the other DBMSs. This situation is somewhat similar to the case of a distributed database management system, where parts of the database reside at different sites that are connected by a communication network.

To maintain the atomicity of multidatabase transaction, it is necessary to have a two-level recovery mechanism. A global recovery manager, or coordinator, is needed in addition to the local recovery managers. The coordinator usually follows a protocol called the two-phase commit protocol, whose two phases can be stated as follows.

PHASE 1: When all participating databases signal the coordinator that the part of the multidatabase transaction involving them has concluded, the coordinator sends a message "prepare for commit" to each participant to get ready for committing the transaction. Each participating database receiving that message will force-write all log records to disk and then send a "ready to commit" or "OK" signal to the coordinator. If the force-writing to disk fails or the local transaction cannot commit for some reason, the participating database sends a "cannot commit" or "not OK" signal to the coordinator. If the coordinator does not receive a reply from a database within a certain time interval, it assumes a "not OK" response.

PHASE 2: If all participating databases reply "OK", the transaction is successful, and the coordinator sends a "commit" signal for the transaction to the participating databases. Because all the local effects of the transaction have been recorded in the logs of the participating databases, recovery from failure is now possible. Each participating database completes transaction commit by writing a commit(T) entry for the transaction in the log and permanently updating the database if needed. On the other hand, if one or more of the participating databases have a "not OK" response to the coordinator, the transaction has failed, and the coordinator sends a message to "roll back" or UNDO the local effect of the transaction to each participating database. This is done

by undoing the transaction operations, using the log.

The net effect of the two-phase commit protocol is that either all participating databases commit the effect of the transaction or none of them do. In case any of the participants – or the coordinator – fails, it is always possible to recover to a state where either the transaction is committed or it is rolled back. A failure during or before phase 1 usually requires the transaction to be rolled back, whereas a failure during phase 2 means that a successful transaction can recover and commit.

Review question 5

Describe the two-phase commit protocol for multidatabase transactions.

Additional content and exercise

Shadow paging

In the shadow page scheme, the database is not directly modified but a copy, stored on permanent storage (e.g. disk), is made of the portion of the database to be modified and all modifications are made to this copy. Meanwhile, the old version of the database remains intact. Once the transaction commits, the modified copy replaces the original in an atomic manner, i.e. the replacement is carried out in its entirety or not at all. If a system crashes at this point, the old version is still available for recovery.

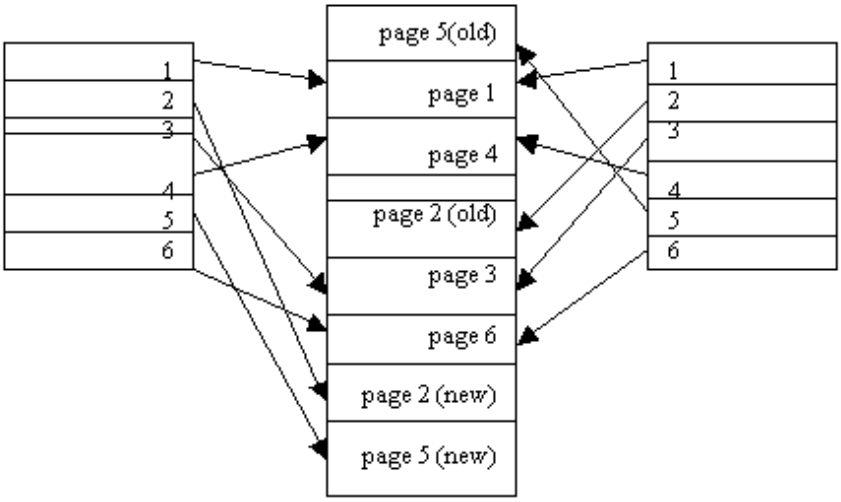
Page management

Before we discuss this scheme, a review of the paging scheme as used in the operating system for virtual memory management is appropriate. The memory that is accessed by a process (a program in execution is a process) is called virtual memory. Virtual memory is divided into pages that are all of a certain size (commonly 4096 bytes or 4K). The virtual or logical pages are mapped onto physical memory blocks (i.e. disk physical records) of the same size as the pages. The mapping is achieved by consulting a page table (or directory). The page table lists each logical page identifier and the address of the physical blocks that actually stores the logical page. The advantage of this scheme is that the consecutive logical pages need not be mapped onto consecutive physical blocks.

Shadow paging scheme considers the database to be made up of a number of fixed-size disk pages (or disk blocks) – say, n – for recovery purposes. A page table (or directory) with n entries is constructed, where the ith page table entry points to the ith database page on disk. The page table is kept in main memory if it is not too large, and all references – reads or writes – to database pages on disk go through the page table.

Shadow paging scheme in a single-user environment

In the shadow page scheme, two page tables are used. The original page table (shadow page table) and the current page table. Initially, both page tables point to the same blocks of physical storage. The current page table is the only route through which a transaction can gain access to the data stored on disk. That is, a transaction always uses the current page table to retrieve the appropriate database blocks.



During transaction execution, the shadow page table is never modified. When a write operation is performed, a new copy of the modified database page is created, but the old copy of that page is not overwritten. Instead, the new page is written elsewhere – on some previously unused disk block. The current page table entry is modified to point to the new disk block, whereas the shadow page table is not modified and continues to point to the old unmodified disk block. The diagram above illustrates the concepts of a shadow page table and a current page table. For pages updated by the transaction, two versions are kept. The old version is referenced by the shadow page table and the new version by the current page table.

To recover from a failure during transaction execution, it is sufficient to free the modified database pages and to discard the current page table. The state of the database before transaction execution is available through the shadow page table, and that state is recovered by reinstating the shadow page table so that it becomes the current page table once more. The database thus is returned to its state prior to the transaction that was executing when the crash occurred,

and any modified pages are discarded. Committing a transaction corresponds to discarding the previous shadow page table and freeing old page tables that it references. Since recovery involves neither undoing nor redoing data items, this technique is called the NO-UNDO/NO-REDO recovery technique.

The advantage of shadow paging is that it makes undoing the effect of the executing transaction very simple. There is no need to undo or redo any transaction operations. In a multi-user environment with concurrent transactions, logs and checkpoints must be incorporated into the shadow paging technique. One disadvantage of shadow paging is that the updated database pages change location on disk. This makes it difficult to keep related database pages close together on disk without complex storage management strategies. Furthermore, if the page table (directory) is large, the overhead of writing shadow page tables to disk as transactions commit is significant. A further complication is how to handle garbage collection when a transaction commits. The old pages referenced by the shadow page that has been updated must be released and added to a list of free pages for future use. These pages are no longer needed after the transaction commits, and the current page table replaces the shadow page table to become the valid page table.

Extension exercise 1: Shadow paging

What is a current page table and a shadow page table? Discuss the advantages and disadvantages of the shadow paging recovery scheme.