Backup and Recovery-I

Hi! In this chapter I am going to discuss with you about Backup and Recovery.

INTRODUCTION

An enterprise's database is a very valuable asset. It is therefore essential that a DBMS provide adequate mechanisms for reducing the chances of a database failure and suitable procedures for recovery when the system does fail due to a software or a hardware problem. These procedures are called *recovery techniques*.

In this chapter, as before, we assume that we are dealing with a multiple user database system and not a single user system that are commonly used on personal computers. The problems of recovery are much simpler in a single user system than in a multiple user system.

Before we discuss causes of database failures and the techniques to recover from them, we should note that in our discussion in this chapter we shall assume that the database is resident on a disk and is transferred from the disk to the main memory when accessed. Modifications of data are initially made in the main memory and later propagated to the disk. Disk storage is often called *nonvolatile storage* because the information stored on a disk survives system crashes like processor and power failure. Non-volatile storage may be destroyed by errors like a head crash but that happens only infrequently. A part of the database needs to be resident in the main memory. Main memory is *volatile storage* because the contents of such storage are lost with a power failure. Given that a database on disk may also be lost, it is desirable to have a copy of the database stored in what is called *stable storage*. Stable storage is assumed to be reliable and unlikely to lose information with any type of system failures. No media like disk or tape can be considered stable, only if several independent nonvolatile storage media are used to replicate information can we achieve something close to a stable storage.

A database system failure may occur due to:

1. *power failures* -- perhaps the most common cause of failure. These result in the loss of the information in the main memory.
2. *operating system or DBMS failure* -- this often results in the loss of the information in the main memory.
3. *user input errors* -- these may lead to an inconsistent data base.
4. *hardware failure (including disk head crash)* -- hardware failures generally result in the loss of the information in the main memory while some hardware failures may result in the loss of the information on the disk as well.
5. *other causes like operator errors, fire, flood, etc.* -- some of these disasters can lead to loss of all information stored in a computer installation.

A system failure may be classified as *soft-fail* or a *hard-fail*. Soft-fail is a more common failure involving only the loss of information in the volatile storage. A hard-fail usually occurs less frequently but is harder to recover from since it may involve loss or corruption of information stored on the disk.

Definition - Failure

A failure of a DBMS occurs when the database system does not meet its specification i.e. the database is in an inconsistent state.

Definition - Recovery

Recovery is the restoration of the database, after a failure, to a consistent state.

Recovery from a failure is possible only if the DBMS maintains redundant data (called *recovery data*) including data about what users have been doing. To be able to recover from failures like a disk head crash, a backup copy of the database must be kept in a safe place. A proper design of recovery system would be based on clear assumptions of the types of failures expected and the probabilities of those failures occurring.

Of course we assume that the recovery process will restore the database to a consistent state as close to the time of failure as possible but this depends on the type of failure and the type of recovery information that the database maintains. If for example the disk on which the database was stored is damaged, the recovery process can only restore the database to the state it was when the database was last archived.

Most modern database systems are able to recover from a soft-fail by a system restart which takes only a few seconds or minutes. A hard-fail requires rebuilding of the database on the disk using one or more backup copies. This may take minutes or even hours depending on the type of the failure and the size of the database. We should note that no set of recovery procedures can cope with all failures and there will always be situations when complete recovery may be impossible. This may happen if the failure results in corruption of some part of the database as well as loss or corruption of redundant data that has been saved for recovery.

We note that recovery requirements in different database environments are likely to be different. For example, in banking or in an airline reservation system it would be a requirement that the possibility of losing information due to a system failure be made very very small. On the other hand, loss of information in a inventory database may not be quite so critical. Very important databases may require fault-tolerant hardware that may for example involve maintaining one or more duplicate copies of the database coupled with suitable recovery procedures.

The component of DBMS that deals with recovery is often called the *recovery manager*.

THE CONCEPT OF A TRANSACTION

Before we discuss recovery procedures, we need to define the concept of a *transaction*. A transaction may be defined as a logical unit of work which may involve a sequence of steps but which normally will be considered by the user as one action. For example, transferring an employee from Department A to Department B may involve updating several relations in a database but would be considered a single transaction. Transactions are straight-line programs devoid of control structures. The sequence of steps in a transaction may lead to inconsistent database temporarily but at the end of the transaction, the database is in a consistent state. It is assumed that a transaction always carries out correct manipulations of the database.

The concept of transaction is important since the user considers them as one unit and assumes that a transaction will be executed in isolation from all other transactions running concurrently that might interfere with the transaction. We must therefore require that actions within a transaction be carried out in a prespecified serial order and in full and if all the actions do not complete successfully for some reason then the partial effects must be undone. A transaction that successfully completes all the actions is said to *commit*. It otherwise aborts and must be *rolled back*. For example, when a transaction involves transferring an employee from Department A to Department B, it would not be acceptable if the result of a failed transaction was that the employee was deleted from Department A but not added to Department B. Haerder and Reuter (1983) summarise the properties of a transaction as follows:

(a) **Atomicity** - although a transaction is conceptually atomic, a transaction would usually consist of a number of steps. It is necessary to make sure that other transactions do not see partial results of a transaction and therefore either all actions of a transaction are completed or the transaction has no effect on the database. Therefore a transaction is either completed successfully or rolled back. This is sometime called *all-or-nothing*.

(b) **Consistency** - although a database may become inconsistent during the execution of a transaction, it is assumed that a completed transaction preserves the consistency of the database.

(c) **Isolation** - as noted earlier, no other transactions should view any partial results of the actions of a transaction since intermediate states may violate consistency. Each transaction must be executed as if it was the only transaction being carried out.

(d) **Durability** - once the transaction has been completed successfully, its effects must persist and a transaction must complete before its effects can be made permanent. A committed transaction cannot be aborted. Also a transaction would have seen the effects of other transactions. We assume those transactions have been committed before the present transaction commits.

When an application program specifies a transaction we will assume that it is specified in the following format:

Begin Transaction
(details of the transaction)
Commit

All the actions between the Begin and Commit are now considered part of a single transaction. If any of these actions fail, all the actions carried out before would need to be undone. We assume that transactions are not nested.

We will use the classical example of a bank account transfer transaction to illustrate some of the concepts used in recovery procedures. The transactions is:

Begin Transaction
Transfer 100 from Account A to Account B
Commit

It is clear that there needs to be a transfer program in the system that will execute the transaction. There are at least the following steps involved in carrying out the transaction:

(a) Read Account A from Disk
(b) If balance is less then 100, return with an appropriate message.
(c) Subtract 100 from balance of Account A.
(d) Write Account A back to Disk.
(e) Read Account B from Disk.
(f) Add 100 to balance of Account B.
(g) Write Account B back to Disk.

We have ignored several small details; for example, we do not check if accounts A and B exist. Also we have assumed that the application program has the authority to access the accounts and transfer the money.

In the above algorithm, a system failure at step (e) or (f) would leave the database in an inconsistent state. This would result in 100 been subtracted from Account A and not added to Account B. A recovery from such failure would normally involve the incomplete transaction being rolled back.

In the discussion above we ignored how the disk manager transfers blocks from and to the disk. For example, in step (1) above, the block containing information on account A may already be in main memory buffers and then a read from disk is not required. More importantly, in Step (d), the write may not result in writing to the disk if the disk manager decides to modify the buffer but not write to disk. Also, in some situations, a write may first lead to a read if the block being modified is not resident in the main memory.

Now that the basic concept of transaction has been described, we can classify the different failures modes that we have discussed earlier.

TYPES OF FAILURES

At the beginning of this chapter we discussed a number of failure modes. Gray *et al* (1981) classifies these failures in the following four classes:

1. **Transaction failure** - a transaction may fail to complete for a number of reasons. For example, a user may cancel the transaction before it completes or the user may enter erroneous data or the DBMS may instruct the transaction to be abandoned and rolled back because of a deadlock or an arithmetic divide by zero or an overflow. Normally a transaction failure does not involve the loss of the contents of the disk or the main memory storage and recovery procedure only involves undoing changes caused by the failed transaction.

2. **System failure** - most of these failures are due to hardware, database management system or operating system failures. A system failure results in the loss of the contents of the main memory but usually does not affect the disk storage. Recovery from system failure therefore involves reconstructing the database using the recovery information saved on the disk.
3. **Media failure** - media failures are failures that result in the loss of some or all the information stored on the disk. Such failures can occur due to hardware failures, for example disk head crash or disk dropped on the floor, or by software failures, for example, bugs in the disk writing routines in the operating system. Such failures can be recovered only if the archive version of the database plus a log of activities since the time of the archive are available.
4. **Unrecoverable Failures** - these are failures that result in loss of data that cannot be recovered and happen usually because of operations errors, for example, failure to make regular archive copies of the database or disasters like an earthquake or a flood.

Gray *et al* (1981) notes that in their experience 97 per cent of all transactions were found to execute successfully. Most of the remaining 3 per cent failed because of incorrect user input or user cancellation. All system crashes were found to occur every few days and almost all of these crashes were due to hardware or operating system failures. Several times a year, the integrity of the disk was lost.

THE CONCEPT OF A LOG

We now discuss some techniques of rolling back a partially completed transaction when a failure has occurred during its execution. It is clear that to be able to roll back a transaction we must store information about what that transaction has done so far. This is usually done by keeping a *log* or a *journal* although techniques that do not use a log exist. A log is an abstraction used by the recovery manager to store information needed to implement the atomicity and durable properties of transactions. The *log manager* is the component of a DBMS that implements the log abstraction. Logs are logically an append-only sequence of unstructured records stored on disk to which an insert is generated at each insert, delete and update. A log can therefore become very large quickly and may become a system performance problem. Each record appended to the log is assigned a unique *log sequence number* for identification.

When a transaction failure occurs, the transaction could be in one of the following situations:

(a) the database was not modified at all and therefore no roll back is necessary. The transaction could be resubmitted if required.

(b) the database was modified but the transaction was not completed. In this case the transaction must be rolled back and may be resubmitted if required.

(c) the database was modified and the transaction was completed but it is not clear if all the modifications were written to the disk. In this case there is a need to ensure that all updates carried out by the completed transaction are durable.

To ensure that the uncompleted transactions can be rolled back and the completed transactions are durable, most recovery mechanisms require the maintenance of a log on a non-volatile medium. Without the log or some other similar technique it would not be possible to discover whether a transaction updated any items in the database. The log maintains a record of all the changes that are made to the database, although different recovery methods that use a log may require the maintenance of somewhat different log. Typically, a log includes entries of the following type:

1. Transaction Number 12345 has begun.
2. Transaction Number 12345 has written x that had old value 1000 and new value 2000.
3. Transaction Number 12345 has committed.
4. Transaction Number 12345 has aborted.

Once a transaction writes commit to the log, it is assumed to commit even if all changes have not been propoagated to the disk. It should however be noted that when an entry is made to the log, the log record may not be immediately written to the disk since normally a system will only write a log block when it is full. A system crash therefore may result in some part of the log that was not yet written to disk also being lost but this does not create any problems. Most systems insist that log blocks be forced out at appropriate times.

If the log maintains information about the database before the updates and after the updates, the information in the log may be used to undo changes when a failure occurs and redo changes that a committed trasaction has made to the database that may have not been written to the disk. Another approach is possible in which the log only maintains information about the database after the updates but the changes are made to the database only if the transaction is completed successfully. We consider both techniques.

Recovery is often a significant cost of maintaining a database system. A major component of the recovery cost is the cost of maintaining the log which is needed only when a crash occurs.

We note that log is the complete history of the database and the maintenance of a log is sometimes needed for auditing purposes and the log then is used for both auditing and recovery. The log may also be used for performance analysis. If a database contained sensitive data, it may be necessary to maintain a log of what information was read by whom and what was written back. This could then be used for auditing the use of the system. This is sometimes called an *audit-trail*. We do not discuss audit aspect of log maintenance any further. Since log is the complete history of the database, access to the log should be carefully controlled. Read access to a log could provide a user indirect access to the whole database.

At recovery time we cannot normally look at all the log since the last failure since the log might be very very big. Markers are therefore often put on the log to ensure that not all the log needs to be looked at when a failure occurs. We will discuss this further later.

Before we discuss the recovery techniques it is necessary to briefly discuss buffer management.

RECOVERY AND BUFFER MANAGEMENT

The part of main memory that is available for storage of copies of disk blocks is called the *buffer*. A software usually is needed to manage the buffer. The primary purpose of a buffer management strategy is to keep the number of disk accesses to a minimum. An easy way to reduce the disk traffic is to allocate a large fraction of the primary memory to storing blocks from disks but there clearly is a limit to how much primary storage can be assigned to buffers. Buffer manager is a specialised virtual memory manager since needs of a database are somewhat more specialised than an ordinary operating system.

1. Replacement strategy
2. Pinned blocks
3. Forced output of blocks

[See CACM July 1881 p 412]

Buffer manager interacts with the crash recovery system to decide on what can be written back to disk. Crash recovery system may insist that some other blocks be forced output before a particular buffer can be output.

To understand the various recovery mechanisms thoroughly, it is essential that we understand the buffer management strategies used in a DBMS. An operating system uses algorithms like the LRU (least recently used) and MRU (most recently used) in managing virtual memory. Similar strategies can be used in database buffer management but a database management does have some peculiar requirements that make some techniques that are suitable for operating systems unsuitable for database management.

Given that the main memory of the computer is volatile, any failure is likely to result in the loss of its contents which often includes updated blocks of data as well as output buffers to the log file.

[Does it need to be rewritten???] All log based recovery algorithms follow the *write-ahead* log principle. This involves writing the recovery data for a transaction to the log before a transaction commits and recovery data for a recoverable data page must be written to the log before the page is written from main memory to the disk. To enforce the write-ahead principle, the recovery manager needs to be integrated with the buffer manager. Some recovery algorithms permit the buffer manager to *steal* dirty main memory pages (a *dirty page* is a page in which data has been updated by a transaction that has not been committed) by writing them to disk before the transactions that modify them commit. The write-ahead principle requires the log records referring to the dirty pages must be written before the pages are cleaned. *No-steal* buffer management policies result in simpler recovery algorithms but limit the length of the transactions. Some recovery algorithms require the buffer manager to *force* all pages modified by a transaction to disk when the transaction commits. Recovery algorithms with *no-force* buffer management policies are more complicated, but they do less I/O than those which force buffers at commit.

We are now ready to discuss recovery techniques which will be in next lecture

REFERENCES

1. R. Bayer and P. Schlichtiger (1984), "Data Management Support for Database Management", Acta Informatica, 21, pp. 1-28.
2. P. Bernstein, N. Goodman and V. Hadzilacos (1987), "Concurrency Control and Recovery in Database Systems", Addison Wesley Pub Co.
3. R. A. Crus (1984), ???, IBM Sys Journal, 1984, No 2.

4. T. Haerder and A. Reuter "Principles of Transaction-Oriented Database Recovery" ACM Computing Survey, Vol 15, Dec 1983, pp 287-318.
5. J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu and I. Traiger (1981), "The Recovery Manager of the System R Database Manager", ACM Computing Surveys, Vol 13, June 1981, pp 223-242
6. J. Gray (1978?), "Notes on Data Base Operating Systems", in Lecture Notes on Computer Science, Volume 60, R. Bayer, R.N. Graham, and G. Seegmueller, Eds., Springer Verlag.
7. J. Gray (1981?), "The Transaction Concept: The Virtues and Limitations" in Proc. of the the 7th International Conf on VLDB, Cannes, France, pp. 144-154.
8. J. Kent, H. Garcia-Molina and J. Chung (1985), "An Experimental Evaluation of Crash Recovery Mechanisms", ACM-SIGMOD??, pp. 113-122.
9. J.S.M. Verhofstad "Recovery Techniques for Database Systems", ACM Computing Surveys, Vol 10, Dec 78, pp 167-196.
10. Mohan???

EXERCISES

1. Log-based recovery techniques use protocols that include

   (a) Do
   (b) Redo
   (c) Undo

2. A database failure may involve the loss of information in the

   (a) main memory
   (b) disk
   (c) memory cache
   (d) all of the above

3. To facilitate recovery, a recovery manager may force buffer manager to

(a) pin some pages

(b)(force some pages

(c) read some pages

(d) write some pages

4. All log-based recovery techniques

   (a) write-ahead log

   (b) use checkpoint or a similar technique

   (c) use the order of the log records to UNDO and REDO

   (d) write log to disk after each transaction update

5. The primary reason why checkpoints are needed in the log is to

   (a) reduce log processing

   (b) maintain consistency

   (c) maintain transaction durability

   (d) reduce database archiving

6. Immediate update may be called logging only the UNDO information.

   Immediate update relinquishes the opportunity to delay writes.

   Immediate update requires that all updated pages be flushed out on transaction commit.

   Immediate update involves updating the database on the disk immediately followed by logging in the update.

7. Deferred update may be called logging only the REDO information.

   Deferred update allows delayed writes.

Deferred update requires that modified pages in memory be pinned in the main memory until the transaction commits.

Deferred update does not need a UNDO procedure.

Deferred update can lead to problems when a transaction requires a large number of pages.

8. Recovery techniques enable recovery from all failures (??)

   A recovery system should be based on clear assumptions about types of failures and probabilities of these failures occurring.

   All recovery techniques require a maintenance of a loop.

9. A transaction is the smallest unit of work.

   A transaction is a logical unit of work which may involve a sequence of steps which preserves the consistency of the database.

   Transactions are not nested.

   Transactions are usually serial programs devoid of loops.

10. UNDO-REDO-DO protocol involves that

    UNDO, REDO (and DO?) be idempotent.

    Only recent log records are looked at.

    UNDO is done backwards and REDO forwards.

    REDO is necessary because all updates of committed transactions may not have been propagated to disk.

    Failures during recovery do not lead to any problems.

11. Which of the following is not true about deferred updates:

    (a) the log may not be written until the transaction commits
    (b) the log must be force-written to disk before any changes are made to the database on the disk

12. Recovery involves the following steps: