Backup and Recovery-II

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

A database might be left in an inconsistent state when:

- deadlock has occurred.
- a transaction aborts after updating the database.
- software or hardware errors.
- incorrect updates have been applied to the database.

If the database is in an inconsistent state, it is necessary to recover to a consistent state. The basis of recovery is to have backups of the data in the database.

Recovery: the dump

The simplest backup technique is `the Dump'.

- entire contents of the database is backed up to an auxiliary store.
- must be performed when the state of the database is consistent - therefore no transactions which modify the database can be running
- dumping can take a long time to perform
- you need to store the data in the database twice.
- as dumping is expensive, it probably cannot be performed as often as one would like.
- a cut-down version can be used to take `snapshots' of the most volatile areas.

Recovery: the transaction log

A technique often used to perform recovery is the transaction log or journal.

- records information about the progress of transactions in a log since the last consistent state.
- the database therefore knows the state of the database before and after each transaction.
- every so often database is returned to a consistent state and the log may be truncated to remove committed transactions.
- when the database is returned to a consistent state the process is often referred to as `checkpointing'.

RECOVERY TECHNIQUES

Consider the following sequence of transactions:

Time	T_1	T_2	T_3
0	Begin	-	-
1	Read N	-	Begin
2	-	-	Read P
3	N := N-5	-	-
4	-	Begin	P := P-4
5	Write N	Read M	-
6	Commit	-	Write P
7	-	Write M	-
8	-	M := M*1.5	Commit
9	-	Temp := M	-
10	-	B := B+1	-

Now assume that a system failure occurs at time 9. Since transactions T_1 and T_3 have committed before the crash, the recovery procedures should ensure that the effects of these transactions are reflected in the database. On the other hand T_2 did not commit and it is the responsibility of the recovery procedure to ensure that when the system restarts the effects of partially completed T_2 are undone for ever.

Jim Gray presented the following three protocols that are needed by the recovery procedures to deal with various recovery situations:

1. DO(T_i, action) - this procedure carries out the action specified. The log is written before an update is carried out.
2. UNDO(T_i) - this procedure undoes the actions of the transaction T_i using the information in the log.
3. REDO(T_i) - this procedure redoes the actions of the committed transaction T_i using the information in the log.

The above actions are sometime called *DO-UNDO-REDO* protocol. They play a slightly different role in the two recovery techniques that we discuss.

When a transaction aborts, the information contained in the log is used to undo the transaction's effects on the database. Most logging disciplines require that a transaction be undone in reverse of the order that its operations were performed. To facilitate this, many logs are maintained as a singly linked list of log records.

As discussed earlier, a system crash divides transactions into three classes. First, there are transactions that committed before the crash. The effects of these transactions must appear in the database after the recovery. If the recovery manager forces all dirty pages to disk at transaction commit time, there is no work necessary to redo committed transactions. Often however, a no-force buffer management policy is used and it is therefore necessary to redo some of the transactions during recovery. The second class of transactions includes those that were aborted before the crash and those that are aborted by the crash. Effects of these transactions must be removed from the database. Another class of transactions is those that were completed but did not commit before the crash. It is usually possible to commit such transactions during recovery.

We now discuss two recovery techniques that use a log. Another technique, that does not use a log, is discussed later.

Immediate Updates

As noted earlier, one possible recovery technique is to allow each transaction to make changes to the database as the transaction is executed and maintain a log of these changes. This is sometimes called *logging only the UNDO information*. The information written in the log would be identifiers of items updated and their old values. For a deleted item, the log would maintain the identifier of the item and its last value. As discussed above, the log must be written to the disk before the updated data item is written back.

We consider a simple example of bank account transfer discussed earlier. When the transaction is ready to be executed, the transaction number is written to the log to indicate the beginning of a transaction, possibly as follows:

<T = 1235, BEGIN>

This is then followed by the write commands, for example the log entries might be:

<T = 1235, Write Account A, old value = 1000, new value = 900>
<T = 1235, Write Account B, old value = 2000, new value = 2100>
<T = 1235, COMMIT>

Our brief discussion above has left several questions unanswered, for example:

(a) Is the write-ahead principle necessary?
(b) If a crash occurs, how does recovery take place?
(c) What happens if a failure occurs as the log is being written to the disk?
(d) What happens if another crash occurs during the recovery procedure?

We now attempt to answer these questions.

If the updated data item is written back to disk before the log is, a crash between updating the database and writing a log would lead to problems since the log would have no record of the update. Since there would be no record, the update could not be undone if required. If the log is written before the actual update item is written to disk, this problem is overcome. Also if a failure occurs during the writing of the log, no damage would have been done since the database update would not have yet taken place. A transaction would be regarded as having committed only if the logging of its updates has been completed, even if all data updates have not been written to the disk.

We now discuss how the recovery takes place on failure. Assuming that the failure results in the loss of the volatile memory only, we are left with the database that may not be consistent (since some transformations that did not commit may have modified the database) and a log file. It is now the responsibility of the recovery procedure to make sure that those transactions that have log records stating that they were committed are in fact committed (some of the changes that such transactions made may not yet have been propagated to the database on the disk). Also effects of partially completed transactions i.e. transactions for which no commit log record exists, are undone. To achieve this, the recovery manager inspects the log when the system restarts and REDOes the transactions that were committed recently (we will later discuss how to identify transactions that are recent) and UNDOes the transactions that were not committed. This rolling back of partially completed transactions requires that the log of the uncommitted transactions be read backwards and each action be undone. The reason for undoing the uncommited transaction backwards becomes if we consider an example in which a data item is updated more than once as follows.

1. **< T = 1135, BEGIN >**
2. **< T = 1135, Write Account A, 1000, 900 >**
3. **< T = 1135, Write Account A, 900, 700 >**
4. **< T = 1135, Write Account B, 2000, 2100 >**

Only backward UNDOing can ensure that the original value of 1000 is restored.

Let us consider a simple example of the log.

1. **< T = 1235, BEGIN >**
2. **< T = 1235, Write Account A, 1000, 900 >**
3. **< T = 1235, Write Account B, 2000, 2100 >**
4. **< T = 1235 COMMIT >**
5. **< T = 1240, BEGIN >**
6. **< T = 1240, Write Account C, 2500, 2000 >**
7. **< T = 1240, Write Account D, 1500, 2000 >**
8. **< T = 1240, COMMIT >**
9. **< T = 1245, BEGIN >**

Let the log consist of the 9 records above at failure. The recovery procedure first checks what transactions have been committed (transactions 1235 and 1240) and REDOes them by the following actions:

REDO (T = 1235)
REDO (T = 1240)

Transaction 1245 was not committed. UNDO (T = 1245) is therefore issued. In the present example, transaction 1245 does not need to be rolled back since it didn't carry out any updates.

The reason for REDOing committed transactions is that although the log states that these transactions were committed, the log is written before the actual updated data items are written to disk and it is just possible that an update carried out by a committed transaction was not written to disk. If the update was in fact written, the REDO must make sure that REDOing the transaction does not have any further effect on the database. Also several failures may occur in a short span of time and we may REDO transactions 1235 and 1240 and UNDO transaction 1245 several times. It is therefore essential that UNDO and REDO operations be *idempotent*; that is, the effect of doing the same UNDO or REDO operations several times is the same as doing it once. An operation like add $35 to the account obviously is not idempotent while an operation setting the account balance to $350 obviously is.

Let us consider another example of UNDO. Let the log have only the first seven records at failure. We therefore issue

REDO (T = 1235)
UNDO (T = 1240)

Once REDO is completed, UNDO reads the log backwards. It reads record of the log and restores the old value of Account D, then reads record 6 of the log and restores the old values of Account C.

Note that the recovery procedure works properly even if there is a failure during recovery since a restart after such a failure would result in the same process being repeated again

Deferred Updates

The immediate update scheme requires that before-update values of the data items that are updated be logged since updates carried out by the transactions that have not been committed may have to be undone. In the deferred updates scheme, the database on the disk is not modified until the transaction reaches its commit point (although all modifications are logged as they are carried out in the main memory) and therefore the before-update values do not need to be logged. Only the after-update data values of the database are recorded on the log as the transaction continues its execution but once the transaction completes successfully (called a *partial commit*) and writes to the log that is ready to commit, the log is force-written to disk and all the writes recorded on the log are then carried out. The transaction then commits. This is sometimes called as *logging only the REDO information*. The technique allows database writes to be delayed but involves forcing the buffers that hold transaction modifications to be pinned in the main memory until the transaction commits.

There are several advantages of the deferred updates procedure. If a system failure occurs before a partial commit, no action is necessary whether the transaction logged any updates or not. A part of the log that was in the main memory may be lost but this does not create any problems. If a partial commit had taken place and the log had been written to the disk then a system failure would have virtually no impact since the deferred writes would be REDOne by the recovery procedure when the system restarts.

The deferred update method can also lead to some difficulties. If the system buffer capacity is limited, there is the possibility that the buffers that have been modified by an uncommitted transaction are forced out to disk by the buffer management part of the DBMS. To ensure that the buffer management will not force any of the uncommitted updates to the disk requires that the buffer must be large enough to hold all the blocks of the database that are required by the active transactions. We again consider the bank account transfer example considered earlier. When the transaction is ready to be executed, the transaction number is written on the log to indicate the beginning of a transaction, possibly as follows:

<T = 1235, BEGIN>

This is then followed by the write commands in the log, for example:

<Write Account A, new value = 900>
<Write Account B, new value = 2100>
<COMMIT T = 1235>

Note that the deferred update only requires the new values to be saved since recovery involves only REDOs and never any UNDOs. Once this log is written to the disk, the updates may be carried out. Should a failure occur at any stage after writing the log and before updating the database on the disk, the log would be used to REDO the transaction.

Similar to the technique used in the last section, the recovery manager needs a procedure REDO which REDOes all transactions that were active recently and committed. No recovery action needs to be taken about transactions that did not partially commit (those transactions for which the log does not have a commit entry) before the failure because the system would not have updated the database on the disk for those transactions. Consider the following example:

1. **< T = 1111, BEGIN >**
2. **< Write BEGIN > ??**
3. **< T = 1111, COMMIT >**
4. **< T = 1235, BEGIN >**
5. **< Write Account A, 900 >**
6. **< Write Account B, 2100 >**
7. **< T = 1235 COMMIT >**
8. **< T = 1240, BEGIN >**
9. **< T = 1240, Write Account C, 2000 >**

10. < T = 1240, Write Account D, 2000 >

11. < T = 1240, COMMIT >

12. < T = 1245, BEGIN >

Now if a failure occurs and the log has the above entries, the recovery manager identifies all transactions that were active recently and have been committed (Transaction 1235 and 1240). We assume the recovery manager does not need to worry about transactions that were committed well before the crash (for example, transaction 1111). In the next section, we will discuss how we identify *recent* transactions.

If another failure occurs during the recovery the same transactions may be REDOne again. As noted earlier, the effect of redoing a transaction several times is the same as doing it once.

[chop this para???] If this log is on the disk, it can handle all system failures except the disk crash. When a failure has occurred, the system goes through the log and carries out transactions T = 1235 and T = 1240. Note that in some cases the transactions may be done more than once but this will not create any problems as discussed above.

System Checkpoints

In the discussion above, the Immediate Update method involves REDOing all recently committed transactions and undoing transactions that were not committed while the Deferred Update method only requires REDOing and no UNDOing. We have so far not defined how *recent* transactions are identified. We do so now.

One technique for identifying recent transactions might be to search the entire log and identify recent transactions on some basis. This is of course certain to be very inefficient since the log may be very long. Also, it is likely that most transactions that are selected for REDOing have already written their updates into the database and do not really need to be REDOne. REDOing all these transactions will cause the recovery procedure to be inefficient.

To limit the number of transactions that need to be reprocessed during recovery after a failure, a technique of marking the log is used. The markers are called system *checkpoints* and putting a marker on the log (called taking a checkpoint) consists of the following steps:

(a) Write a <begin checkpoint> record to the log (on the disk?),

(b) all log records currently residing in the main memory are written to the disk followed by the writing to the disk of all modified pages in the buffer. Also identifiers (or names) of all active transactions are written to the log.

(c) Write an <end checkpoint> record to the log.

Now when the recovery procedure is invoked on a failure, the last checkpoint is found (this information is often recorded in a file called the *restart file*) and only the transactions active at the time of checkpoint and those after the checkpoint are processed.

A simple recovery algorithm requires that the recovery manager identify the last checkpoint and builds two lists, one of the transactions that need to be redone and the other of transactions that need to be undone. Initially, all transactions that are listed as active at the checkpoint are included in the UNDO list and the log is scanned forward (show the lists and show the steps???). Any new transaction that becomes active is added to the UNDO list and any transaction that is logged to have committed is removed from the UNDO list and placed on the REDO list. At the end of the log, the log is scanned backwards and all the actions of those transactions that are on the UNDO list are UNDOne in the backward order. Once the checkpoint is reached, the log is scanned forward and all the actions of the transactions on the REDO list are REDOne. (s/a??)

Summary of Log-based Methods

Although log-based recovery methods vary in a number of details, there are a number of common requirements on the recovery log. First, all recovery methods adhere to the write-ahead log procedure. Information is always written to the log before it propagates to non-volatile memory and before transactions commit. Second, all recovery methods rely on the ordering of operations expressed in the log to provide an ordering for the REDO and UNDO operations. Third, all methods use checkpoints or some similar technique to bound the amount of log processed for recovery. Some recovery methods scan the log sequentially forward during crash recovery. Others scan the log sequentially backwards. Still others use both forward and backward scans.

A log manager may manage a log as a file on the secondary storage. When this file has reached a pre-specified length, the log may be switched to another file while the previous log is copied to some archive storage, generally a tape. The approach used by System R is to take a large chunk of disk space and lay it out for the log as a circular buffer of log pages. The log manager appends new blocks sequentially to the circular buffer as the log fills. Although an abstract log is an infinite resource, in practice the online disk space available for storing the log is limited. Some systems will spool truncated log records to tape storage for use in media recovery. Other systems will provide enough online log space for media recovery and will discard truncated log data. (detail??)

Shadow Page Schemes

Not all recovery techniques make use of a log. Shadow page schemes (or *careful replacement* schemes) are recovery techniques that do not use logging for recovery. In these schemes when a page of storage is modified by a transaction a new page is allocated for the modified data and the old page remains as a *shadow* copy of the data. This is easily achieved by maintaining two tables of page addresses, one table called the *current page table* and the other called the *shadow page table*. At the beginning of a transaction the two page tables are identical but as the transaction modifies pages new pages are allocated and the current page table is modified accordingly while the shadow page table continues to point to the old pages. The current pages may be located in the main memory or on the disk but all current pages are output to the disk before the transaction commits. If a failure occurs before the transaction commits, the shadow page table is used to recover the database state before the transaction started. When a transaction commits, the pages in the current page table (which now must be on the disk) become the pages in the shadow page table. The shadow pages must be *carefully replaced* with the new pages in an atomic operation. To achieve this, the current page table is output to the disk after all current pages have been output. Now the address of the shadow page table is replaced by the address of the current page table and the current page table becomes the shadow page table committing the transaction. Should a failure occur before this change, the old shadow page table is used to recover the database state before the transaction.

Shadow page recovery technique eliminates the need for the log although the technique is sometime criticized as having poor performance for normal processing. However, the recovery is often fast when shadow paging is used. Also the technique requires a suitable technique for garbage collection to remove all old shadow pages.

Evaluation

(add more to this section? rewrite it??) Gray et al in their paper note that the recovery system was comparatively easy to write and added about 10 percent to the DBMS code. In addition, the cost of writing log records was typically of the order of 5 percent. Cost of checkpoints was found to be minimal and restart after a crash was found to be quite fast. Different recovery algorithms have varying costs, both during normal processing and recovery. For the log algorithms, the costs of algorithms during normal processing include the volume and frequency of the log writes and the algorithms influence on buffer management. Recovery algorithms which force the buffer pool have comparably poor performance. The costs of crash recovery include the number of log records read, and the number of pages of data that must be paged into main memory, restored and paged out. No-force recovery algorithms will read more data pages during recovery than force algorithms. Recovery algorithms with steal buffer pool management policies may read data pages for both redo and undo processing, while no steal buffer managers mainly read pages for redo processing.

RECOVERING FROM A DISK CRASH (Media failure?)

So far, our discussion about recovery has assumed that a system failure was a soft-failure in that only the contents of the volatile memory were lost and the disk was left intact. Of course, a disk is not immune from failure and a head crash may result in all the contents of the disk being lost. Other, more strange, failures are also possible. For example, a disk pack may be dropped on the floor by a careless operator or there may be some disaster as noted before. We therefore need to be prepared for a disaster involving loss of the contents of the nonvolatile memory.

As noted earlier, the primary technique for recovery from such loss of information is to maintain a suitable back up copy of the database possibly on tape or on a separate disk pack and ensure its safe storage possibly in a fire-proof and water-proof safe preferably at a location some distance away from the database site. Such back up may need to be done every day or may be done less or more frequently depending upon the value the organisation attaches to loss of some information when a disk crash occurs.

Since databases tend to be large, a complete back up is usually quite time consuming. Often then the database may be backed up incrementally i.e. only copies of altered parts of the database are backed up. A complete backup is still needed regularly but it would not need to be done so frequently. After a system failure, the incremental system tapes and the last complete back up tape may be used to restore the database to a past consistent state.

Rollback

The process of undoing changes done to the disk under immediate update is frequently referred to as rollback.

- Where the DBMS does not prevent one transaction from reading uncommitted modifications (a `dirty read') of another transaction (i.e. the uncommitted dependency problem) then aborting the first transaction also means aborting all the transactions which have performed these dirty reads.
- as a transaction is aborted, it can therefore cause aborts in other dirty reader transactions, which in turn can cause other aborts in other dirty reader transaction. This is referred to as `cascade rollback'.

Important Note

Deferred Update

Deferred update, or NO-UNDO/REDO, is an algorithm to support ABORT and machine failure scenarios.

- *While a transaction runs, no changes made by that transaction are recorded in the database.*
- *On a commit:*
- *The new data is recorded in a log file and flushed to disk*
- *The new data is then recorded in the database itself.*
- *On an abort, do nothing (the database has not been changed).*
- *On a system restart after a failure, REDO the log.*

If the DMBS fails and is restarted:

- *The disks are physically or logically damaged then recovery from the log is impossible and instead a restore from a dump is needed.*
- *If the disks are OK then the database consistency must be maintained. Writes to the disk which was in progress at the time of the failure may have only been partially done.*
- *Parse the log file, and where a transaction has been ended with `COMMIT' apply the data part of the log to the database.*
- *If a log entry for a transaction ends with anything other than COMMIT, do nothing for that transaction.*
- *flush the data to the disk, and then truncate the log to zero.*
- [illegible]

Immediate Update

Immediate update, or UNDO/REDO, is another algorithm to support ABORT and machine failure scenarios.

- *While a transaction runs, changes made by that transaction can be written to the database at any time. However, the original and the new data being written must both be stored in the log BEFORE storing it on the database disk.*
- *On a commit:*
- *All the updates which has not yet been recorded on the disk is first stored in the log file and then flushed to disk.*
- *The new data is then recorded in the database itself.*
- *On an abort, REDO all the changes which that transaction has made to the database disk using the*
- *log entries.*
- *On a system restart after a failure, REDO committed changes from log.*

If the DMBS fails and is restarted:

- *The disks are physically or logically damaged then recovery from the log is impossible and instead a restore from a dump is needed.*
- *If the disks are OK then the database consistency must be maintained. Writes to the disk which was in progress at the time of the failure may have only been partially done.*
- *Parse the log file, and where a transaction has been ended with `COMMIT' apply the `new data' part of the log to the database.*
- *If a log entry for a transaction ends with anything other than COMMIT, apply the `old data' part of the log to the database.*
- *flush the data to the disk, and then truncate the log to zero.*

Review Question

1. What is Recovery and why it is important?
2. Explain recovery techniques

EXERCISES

1. Given the log on page 6 of these notes, what transactions need to be REDOne or UNDOne if a failure occurs:

(a) after step 8

(b) after step 10

2. What difference is expected if deferred update procedure is being used?
3. Explain what would happen if a failure occurred during the recovery procedure in Ex. 1.
4. A system crash divides transactions into the following classes:

(a) Committed before the checkpoint

(b) Not completed

(c) Not committed at all

(d) Inactive

5. Give an example of each

(a) transaction failure

(b) system failure

(c) media failure

(d) unrecoverable failure

6. Given that one million transactions are run on a database system each day, how many transactions are likely to fail:

(a) 10,000 to 100,000

(b) 1,000 to 10,000

(c) 100 to 1,000

(d) below 100

7. Which of the following are properties of a transaction

(a) atomicity

(b) consistency

(c) isolation

(d) durability

(e) idempotency

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???