CS352 Lecture - Crash Recovery

Last revised March 4, 2019

Objectives:

1. To introduce the use of a log with deferred update
2. To introduce the use of a log with immediate update
3. To introduce shadow paging

Materials:

1. PROJECTABLES showing Shadow Paging

I. Introduction

A. One of the most important functions of a DBMS is ensuring the integrity of the data in the face of various unpredictable events such as power failures, hardware failures, software failures etc. In fact, the uniform protection of data that a good DBMS provides may be one of its greatest advantages over traditional file processing.

We have seen how a DBMS can enforce the atomicity and isolation properties of an ACID transaction. Today we will look at how it handles the durability property.

B. In general, it is necessary to protect data against several kinds of sources of corruption:

1. Logical errors in the incoming data, which cause an operation to be aborted before it is completed. (Also in this category is the possibility that an interactive system may allow a user to abort an operation he has begun before it completes.)

2. Failure of a transaction to complete execution due to issues related to concurrency (e.g. rollback, deadlock).

3. System crashes that shut down the system unexpectedly. These can arise from:

   a) Power failures.

   b) Hardware failures - e.g. a chip going bad.

   c) Software failures - e.g. operating system crashes.

   d) Network communication failures (due to many possible causes)

   e) Human error - an operator pressing a wrong button or issuing a command that crashes the system.

   If a system crash occurs in the middle of updating the database, the writing of the new data may be only partially completed, resulting in corrupted data.

4. Hardware failures that damage the media storing data. Of these, the most potentially catastrophic is a head crash, in which a disk drive head comes into contact with the surface of the disk - effectively destroying all the data on the platter.

5. External catastrophes such as fire, flood, etc.

C. In general, data is stored in three types of storage, each with its own degree of security against loss:

1. Data in VOLATILE STORAGE - e.g. the main memory of the computer - is subject to loss at any time due to any kind of system failure. In particular, power failures, most hardware failures, and many software crashes will cause data in volatile storage to be lost.

2. Data in NON-VOLATILE STORAGE - e.g. disk and tape - is much more secure. Data in non-volatile storage is generally not lost unless there is a power failure while it is being written or a catastrophic failure of the storage device itself (e.g. a head crash on a disk) or an external catastrophe such as fire or flooding. In this regard, tape is much less vulnerable than disk.

3. Conceptually, STABLE STORAGE is storage that is immune to any kind of loss. While no storage medium is totally immune to destruction of data, stable storage may be approximated in one of two ways:

   a) The use of certain kinds of storage media, such as write-once disks. Data written on a such media is immune to virtually any possible source of damage short of the physical damage to the disk itself.

   b) The writing of the same data on more than one non-volatile medium, so that if one is damaged the other(s) will still retain the data intact.

      (1)Use of on-site mirroring - e.g. through RAID

      (2)Use of off-site mirroring (remote backup), which protects against physical dangers as well as system errors/crashes.

D. Actually, the protection of data involves two different sorts of measures.

1. Regular system backups are an essential part of protecting data against loss. Backups are designed to prevent loss of data due to physical damage to non-volatile storage media (e.g. head crashes on the disk, fire, etc.), and also provide some protection against inadvertent erasure of data that is still needed.

2. Measures taken to recover from aborted operations and system crashes.

   a) Note that backup is particularly designed to protect against loss of data due to damage to non-volatile storage media - a relatively rare

occurrence. Since any work done since the last backup is not saved, other measures need to be taken to allow a fast restoration to normal operation after an aborted operation or a system crash. (It is not generally acceptable for a late afternoon power failure to be able to destroy all work done since the daily backup was run in the morning!)

b) It is these measures we focus on today. The measures we will discuss seek to allow a rapid restoration of the system to a consistent state after an aborted operation or a crash that does NOT result in damage to non-volatile storage media (i.e. only the contents of volatile storage are lost.)

3. The concept of a transaction will be a the heart of our discussion. In particular, the measures we will discuss are related to ensuring the durability property of a transaction. We will assume that each transaction is assigned a unique identification (e.g. a serial number), and that some record of incoming transactions is kept. We must ensure the durability of every transaction that committed before the crash, and must also deal with transactions that were in process at the time of the crash - generally by removing all their effects from the database since they did not commit.

E. Crucial to many schemes for guaranteeing the consistency of a database is the notion of a processing LOG, stored in "stable" storage. During the processing of each transaction, a series of entries are made in the log. Each entry includes the transaction's serial number, an entry type, and possibly other data.

1. When the transaction begins, a start transaction entry is made in the log.

2. Appropriate entries are made in the log to record changes that the transaction makes to the database (more on these later.)

3. One of two types of entry is made in the log when the transaction completes:

a) A COMMIT entry indicates that the transaction completed successfully, so that the durability all its changes to the database should be preserved.

b) An ABORT entry would indicate that the transaction failed for some internal reason (logical error in the data or user abort), so that none of its changes to the database should be allowed to remain.

c) If a transaction was in process (but not finished) when a system crash occurs, then neither of these entries will appear in the log. This implies that:

(1)No changes that the transaction has already made to the database should be allowed to remain (i.e. the effect should be the same as if the transaction aborted.)

(2)But then, if possible, the transaction can be restarted from the beginning after the database is restored to a consistent state.

4. We suggested that the log should be maintained in stable storage. Actually, this is not absolutely necessary. Non-volatile storage can be used for the log (and often is).

a) One way to achieve a measure of stability is to keep two copies of the log on separate media, so that failure of one medium will not destroy the log

b) Ideally, the second copy of the log is kept at a remote site.

c) In either case, however, we DO have to ensure that the log data is written to the storage medium BEFORE the changes it records are actually written.

(1)Since each log entry will be relatively short, the system will normally buffer entries in primary storage until a full block of entries has accumulated, and then will write that block to the log, followed by updating the various database entries on disk.

(2)When we talked about buffer management, we noted that under some circumstances a particular data buffer would have to be pinned so that it can't be written back to disk. One situation where this happens is the case where a change has been made to an in-memory copy of a row, but the corresponding log entry recording the change has not yet been physically written to the log. In this case, the data buffer must be pinned until the log block has been filled up and written.

(3)If, for some reason, it becomes necessary to write out the data block before the log block recording the change has been written, then we will need to FORCE the premature writing of a partially-filled log block

(4)If we are using a second copy of the log at a remote site, then a very safe strategy would prohibit committing a transaction if we cannot write the log entry at BOTH sites due to a communication problem or failure of the remote site. Since this could prevent any work from occurring, this can relaxed to allow work to proceed with some minimal risk of data loss in such cases.

d) Further, we have to ensure that a system crash during the writing of a particular log block cannot corrupt data previously written to the log. This is a particular concern if we have to force the output of a log block before it is full. It would be reasonable to consider reading this block back in, adding more data to it, and then writing it back out. But a failure during this rewrite could corrupt the previous data! So this cannot be allowed.

F. We will consider three general schemes for crash control:

1. Incremental Log with Deferred updates: no changes are actually made to the database until AFTER the transaction has committed and the COMMIT entry has been written to the log.

2. Incremental Log with Immediate updates: changes are made to the database during the processing of the transaction; but only AFTER a log entry is made indicating the original value so that the change can be reversed should the system crash or the transaction fail for any reason.

3. Shadow paging: two complete copies of the relevant database data are kept during the transaction: the original values and the modified values. Only after the transaction commits are the modified values made to permanently replace the original ones. (No log is needed for this strategy, of course)

II. Incremental Log with Deferred Updates

A. In this scheme, no actual changes are made to the database until the transaction completes. Instead, the DBMS converts each write operation issued by the transaction to a write to the LOG, in which it records the item that would be altered and the new value that would be put there.

Example: A transaction to transfer AMOUNT dollars from a user's checking account number CACCOUNT to his savings account number SACCOUNT might execute the following SQL program:

```
update checking_accounts
 set balance = balance - :amount
 where account_no = :caccount;
update savings_accounts
 set balance = balance + :amount
 where account_no = :saccount;
```

If the initial balances in each account are $1000 and $2000 respectively, and the amount to be transferred is $50.00, then the following entries would be

made in the log. (Assume the transaction has been given the serial number T1234, and that the account records in question have internal identifiers (record numbers) 127 and 253.)

t1234 starts
t1234 writes 950.00 to balance of checking_accounts record 127
t1234 writes 2050.00 to balance of savings_accounts record 253
t1234 commits

B. Only when the transaction has partially committed are the actual updates to the database made.

Example: as soon as the T1234 commits record has been written to stable storage, the DBMS would proceed to write the two new values into the database.

C. If the transaction is user-aborted or fails for some reason, no harm is done, since no changes have been made to the database.

D. Should the system crash during the course of execution of the transaction, one of two things will happen:

1. If the crash occurs BEFORE the commits record has been written, then the transaction can either be simply be restarted from the beginning or ignored (the latter if it was an interactive transaction and the system was down long enough for the user to have left.) Either approach will leave the system in a consistent state.

2. If the crash occurs AFTER the commits record has been written, then each of the values specified in the log will be written to the specified locations. Note that the data may already have been written; but no harm is done writing the same value to the same location a second time.

E. Under this scheme, the following algorithm is executed whenever the system restarts after a crash:

for each transaction whose start record appears in the log

```
if its commit record has been written to the log then
     write each of the new values specified in the log to
          the database
// else do nothing
```

1. This is called REDO-ing the transaction. Note that what is redone is not the actual computation, but the writing of the results of the computation which have been preserved in the log.

2. Note that all transactions which have written their commits record must be redone - not just the last one. This is because the actual updates to the database may be made to in-memory buffers - not to the actual disk - depending on what buffering strategy is used. Thus, the only way to ensure that a transaction's effect is consistently reflected in the database after a crash is to redo all of its writes.

3. Because the requirement of redoing all partially-committed transactions after a crash can be burdensome, it is common to make use of CHECKPOINTS.

   a) From time to time (perhaps during slack periods), the DBMS will suspend processing of transactions and flush all its internal buffers to disk. This means that all committed transactions are now fully reflected in non-volatile storage.

   b) The DBMS now writes a CHECKPOINT entry to the log, indicating that all prior committed transactions have been fully entered into non-volatile storage. This entry will also include the number(s) of any transaction(s) currently in process.

   c) If this is done, then when a crash occurs the redoing of transactions can be begun with the first transactions that were active at the time of the most recent checkpoint entry in the log. (It will be necessary to read back in the log past the checkpoint record to the start record(s) of the active transaction(s)).

III. Incremental Log with Incremental Update

A. This scheme differs from the previous scheme in that it actually makes changes to the database during the course of processing a transaction, rather than waiting to do them until after the transaction completes. This requires that more information be placed in the LOG.

1. Each write operation results in an entry to the log, as in the previous scheme. This entry must be written out to stable storage BEFORE the corresponding data is written to the database.

2. The entry includes both the new value and the OLD value.

   Example: For our previous example, the log would now look like this:

   t1234 starts
   t1234 writes 950.00 to balance of checking_accounts record 127
   (old value was 1000.00)
   t1234 writes 2050.00 to balance of savings_accounts record 253
   (old value was 2000.00)
   t1234 commits

B. If the transaction is user-aborted or fails, then all of the writes that have already taken place must be undone by writing the original value back to the database.

C. Should a crash occur, the following must be executed when the system restarts:

```
for each transaction whose start record appears in the log
if its commit record has been written to the log then
     write each of the new values specified in the log to the
        database
else
     rewrite each of the old values specified in the log to
        the database
```

1. As before, writing the new values specified in the log is called REDO-ing the transaction, and may result in writing values that have been already been written (which is harmless.)

2. Writing the old values specified in the log back is called UNDO-ing the transaction. This may also result in writing a value that has already been written back (due to a prior abort of the transaction in question) or in writing a value to a location that has never been changed (because the system crashed before the write recorded in the log actually took place.) In either case, no harm is done.

3. Note, though, that the ORDER of these operations is critical.

   a) All undo operations must be done FIRST, in reverse order (most recent first ...)

   b) All redo operations can then be done, in forward order (oldest first).

      This is necessary to prevent undoing a redo or writing a wrong value in an undo.

D. Checkpointing may be used with this scheme as with the prior one to reduce the number of transactions to be processed during crash recovery.

E. In general, this scheme involves somewhat more overhead than the previous scheme.

   1. Log entries are longer (two values, not just one, must be recorded)

   2. A user-aborted or failed transaction requires work to clean up its effects (not so with the prior scheme.)

   3. After a crash, some action must be taken for EVERY transaction in the log, not just for those that committed.

4. Very important: before every write operation, the log entry must be forced out to stable storage. In the previous scheme, log entries need only be forced out to stable storage at commit time.

F. However, this scheme does allow changes made by a transaction to become visible in the database more quickly. (This is not necessarily always an advantage, however, since it can interfere with recoverability or lead to cascading rollback it the transaction fails. However, one place where this might be useful is a transaction that writes a large amount of data before committing - too much data to buffer in internal memory - e.g.

```
update savings
 set balance = balance + interest_rate * balance;
```

IV. Shadow Paging

A. The final method for handling recover is quite different from the other two, since it does not make use of a log at all. Instead, it works by maintaining two separate versions of the active portion of the database: the current version (which reflects all changes made thus far) and the shadow version (which reflects the state of the database before the transaction started.)

1. A particular transaction "sees" only the current version. In a concurrent environment, other users either "see" the shadow version still, or are prevented from accessing the item at all until the current transaction commits.

2. If a particular transaction is user aborted or fails, then the current version of the database is discarded and the shadow version remains as it was.

3. When the transaction commits, the current version is made permanent by a simple pointer readjustment.

B. Of course, doing this with the entire database could require a huge amount of additional storage, copying of information, etc. The name shadow PAGING comes from the fact that the system views the physical storage as a series of pages (blocks) numbered 1 .. n (where n can be very large). No requirement is imposed that consecutive pages occupy consecutive physical locations on the disk; instead, a page table is used to map page numbers to physical locations.

   PROJECT page table

   Access to the page table is via a pointer to it kept in some known location on disk.

C. As noted in the book, this scheme is a harder to use with concurrent processing than the other schemes - since now each concurrent transaction must have its own private current page table and interaction between transactions is harder to control. For this reason, it is not often used.