CS352 Lecture - Recovery

Last revised April 2, 2021

Objectives:

1. To introduce the use of a log
2. To briefly introduce shadow paging

Materials:

1. Projectable of transaction states
2. Projectable showing need to undo from latest to earliest
3. Projectable of a rollback and effect in log
4. Projectable showing need to redo from earliest to latest
5. Projectable of example of using log for recovery
6. Projectable of recovery algorithm
7. Projectable showing Shadow Paging

I. **Introduction**

A. We have been speaking about the notion of ACID transactions.

1. Consistency is the responsibility of the transaction author - not the DBMS.

2. In our discussion of concurrency, we have seen how a DBMS can enforce the isolation properties of an ACID transaction.

3. Today we will look at how it handles the atomic and durability properties.

B. Recall the notion of the state of a transaction.

PROJECT

1. Since a transaction may start making changes to the database before it commits - and therefore might fail after some changes have already been made.

   Atomicity requires that all effects of a transaction that enters the failed state must be removed from the database. This is initiated by the SQL statement `rollback`

2. It is also possible that the system may crash while a transaction is active for a variety of possible reasons:

   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.

      Again, atomicity requires that, in such cases, all effects of a transaction that was active when a crash occurred must be removed from the database before normal operation can resume. This must be done when the database is restarted after the cause of the crash is rectified.

3. Once a transaction has entered the fully committed state, it is necessary to ensure that changes that it made to the database are durable.

   a) If a system crash occurs, durability requires ensuring that all effects of a transaction that fully committed must remain in the database when normal operation is restored. This is the focus of our discussion today.

b) Durability also requires use of things like mirroring and on-site and off-site backups to deals with some issues that are not our focus today:

(1)Being able to deal with more catastrophic 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.

(2)Being able to deal with physical catastrophes such as fire, flood, etc.

C. Storage Media Issues

1. Recall that data is stored in three types of storage, each with its own degree of security against loss:

a) 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, hardware failures, and most software crashes will cause data in volatile storage to be lost.

b) 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.

c) 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 by 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 over a network which protects against physical dangers as well as system errors/crashes.

(3)The latter approach is also of value in ensuring protection from certain kinds of physical damage to a system such as fire, flood, etc.

2. When we looked at file storage media earlier, we saw that disk and SSD's store data in blocks, such that information is transferred to and from these media as entire blocks.

   a) This means that write operations by a transaction are typically done to an in-memory buffer, with the actual transfer of the data to the disk occurring at a later time when the in-memory buffer is actually written to disk.

   b) Since main memory is volatile, that means that data that was written by a transaction can be lost if some sort of failure occurs before the data is actually written to non-volatile storage, which may not happen until many transactions later.

   c) Thus, the mere fact that a transaction is committed does not guarantee that all changes it has made to the database have actually made it to non-volatile storage.

II.Logging

A. Crucial to guaranteeing the consistency of a database is the notion of a processing LOG, stored in stable storage.

B. During the processing of each transaction, a series of entries are made in the log. When a transaction starts processing, it is assigned a unique serial number. Each entry in the log will includes the transaction's serial number, an entry type, and possibly other data.

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

2. Appropriate entries are made in the log to record changes that the transaction makes to the database. A key principle is that whenever an operation is performed that leads to changes in what is stored in the database, a <write> entry is first made in the log and only then is an actual change to the database initiated.

    a) This is called write-ahead logging.

    b) In the logging approach known as immediate update, in addition to recording the transaction performing the operation, the <write> entry in the log also records the location being changed and the old and new values in that entry.

    c) Often, the location is a particular page in the file, in which case the log entry must record surrounding context as well. For simplicity, though, we will act as if the log just needs to include the one item being changed.

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. This is written when the transaction executes a `commit` operation in SQL, or it commits implicitly through some sort of autocommit.

b) An <abort> entry indicates not only that the transaction failed for but also that all effects of the transaction have been removed from the database - so the state of the database is the same as it would be if the transaction had never occurred. (So SQL `rollback` just puts the transaction in the failed state - the <abort> entry indicates that rolling the transaction back has actually been done.)

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.

4. (We will encounter another type of entry called redo-only shortly.)

C. 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).

1. 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

2. Ideally, the second copy of the log is kept at a remote site accessed over a network.

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

a) 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.

b) If a change has been made to an in-memory copy of a block, but the corresponding log entry recording the change has not yet been physically written to the log, the in-memory buffer for the block must be pinned until the log block has been filled up and written.

c) 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\

d) 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.

4. 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 seem 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.

III.Use of the Log

A. Information in the log is used to ensure atomicity when a transaction enters the failed state.

1. The log is scanned backwards - stopping when the `<start>` entry for the transaction is encountered.

a) Each time a `<write>` entry for the transaction is encountered, the old value recorded in the entry is written back to the database. This ensures atomicity by removing all effects of the transaction from the database.

b) The order in which this is done is important. To see why, consider the log that might be produced by a transaction that writes the same location twice, where the transaction fails just after the second write.

```
<T start>
<T write, A, 0, 1>
<T write, A, 1, 2>
```

PROJECT

If the old values are written back in the the same order in which they appear in the log, what will be written to A? What should it be?

ASK

1 will be written - it should be 0

c) Working backward from the latest entry in the log ensures that the original value in the database when the transaction was started is the one that is restored. (It also means we can stop scanning the log when we reach the `<start>` entry for the transaction.)

2. When this is done, one or more special "redo-only" entries are added to the log as well. This deals with the possibility that a system crash may occur before the rewritten value ends up in non-volatile storage, and also simplifies the crash recovery process.

3. This is referred to as UNDOING a transaction

4. Once all the transaction's effects have been undone - and only then - the <abort> entry can be written.

5. Thus, the following transaction, done with A initially 0,

```
set A to 1
set A to 2
rollback;
```

PROJECT

results in the following log entries, with A restored to 0

```
<T start>
<T write A, 0, 1>
<T write, A, 1, 2>
<T redo-only, A, 1>
<T redo-only, A, 0>
<T abort>
```

PROJECT

B. When a crash occurs, the log is used to ensure atomicity and durability.

1. If a `<commit>` entry for the transaction appears in the log, it indicates that the transaction has full committed,. Information in the log is used to ensure durability following a crash.

a) it is possible that some of its write operations were in volatile storage and had not yet been copied to non-volatile storage. To ensure durability it is necessary to ensure that each of the "new" values in its <write> is actually present on disk. This can be done by simply writing them again, since no harm is done by writing the same value twice.

b) This is referred to as REDOING a transaction. Note that - in this context - to REDO a transaction simply means ensuring that the changes it made to the database are restored if necessary - the computation done by the transaction is not redone - just its effects. When a crash occurs, information in the log is also used to ensure atomicity by undoing any changes recorded by <write> entries that pertain to transactions for which a <start> entry occurred but no <abort> or <commit> entry appears in the log.

c) Again, the order for doing this is important. To see why, consider the log that might be produced by two transactions that each commit after writing the same location, if a crash occurs just after the second commit.

```
<T1 start>
<T1 write, A, 0, 1>
<T1 commit>
<T2 start>
<T2 write, A, 1, 2>
<T2 commit>
```

PROJECT

If the old values are written back in the the same order from earliest to latest (as needs to be the case for undo), what will be written to A? What should it be?

ASK

1 will be written - it should be 2

2. It may appear that, when a crash occurs, transactions for which an `<abort>` entry appears in the log may be ignored, since their effects were already removed from the database.

   a) This is not true, though. If a crash occurs soon after a transaction is aborted, it is possible that some of the rewrites of old values needed to undo it have not yet found their way to non-volatile storage. Therefore, redo-only entries do need to be rewritten, just in case. This can be done by simply writing them again, since no harm is done by writing the same value twice.

   b) It turns out that the algorithm is simplified if the `<write>` entries for this transaction are actually redone - which seems counter-intuitive, since they were just undone! However, no lasting harm is done since the redo-only entries that appear subsequently will wipe their effect out again - and the algorithm is actually simpler this way.

3. When a crash occurs, transactions for which a `<start>` entry appears in the log but no `<commit>` or `<abort>` entry appears were active at the time of the crash. They need to be undone. Since they did not commit, they are treated as if they had failed (though it may be possible to restart the transaction from the beginning.) This means that their `<write>` entries must be undone, `<redo-only>` entries must be added to the log, and an `<abort>` entry for the transaction must be added to the log when recovery is finished.

   This implies, of course, that every transaction except one active when a crash occurs will have either a `<commit>` or an `<abort>` entry in the log, and that a transaction will be totally undone at most once - and after recovery is complete all transactions (even those active at the time of the crash) will either a `<commit>` or an `<abort>` entry in the log.

C. Example:

Suppose, when a crash occurs, that the log contains the following entries:

```
<T1 start>
<T1 write, A, 0, 1>
<T1 commit>
<T2 start>
<T2 write, B, 2, 3>
<T2 redo-only B, 2>
<T2 abort>
<T3 start>
<T3 write, C, 4, 5>
```

PROJECT

1. T1 must be redone, because its <commit> entry appears in the log. Therefore, it is redone, and its new value 1 is written to A just to be sure it is there. (It may be there already, but no harm writing twice).

2. T2 has already been undone, because its <abort> entry appears in the log. Its <redo-only> entry must be redone, ensuring B contains the value 2. The algorithm we will see shortly will also redo its <write>, which will temporarily change B to 3, but that will be replaced when the <redo-only> entry is redone.

3. T3 was active at the time of the crash (its <start> appeared in the log, but no <commit> or <abort>). To do this, the old value of 4 is written to C, and new <T3 redo-only C 4> and <T3 abort> entries are added to the end of the log

D. All of this would imply that, when a crash occurs, myriads of entries in the log might have to be either redone or undone. To reduce the potential this would have for delaying recovery from a crash, it is possible to periodically perform a checkpoint operation.

1. At a checkpoint, all log entries that have not yet been written to stable storage are flushed, and then all write entries are also flushed..

2. A <checkpoint> entry in the log can be added containing a list of transactions that were active at the time of the checkpoint.

3. Should a crash occur before the next checkpoint, only log entries for these transactions or those started after the checkpoint need to be dealt with in the event of a crash.

4. Only the most recent checkpoint must be considered in recovery.

E. This leads to the following recovery algorithm

```
find the most recent <checkpoint> entry
    copy the lists of entries that were active at the
        time of the checkpoint into a list of
        transactions that may need to be undone
scan the entries in the log from the checkpoint forward
    // This is called the redo phase
    if the entry is a <start>
        add the transaction to the list of transactions
            that may need to be undone
    else if the entry is a <commit> or <abort>
        remove the transaction from the list of
            transactions that may need to be undone
    else (it is a <write> or <redo-only)
        redo it
scan the entries in the log backward from the end of the
        log until the list of transactions needing to be
        undone is empty
    // Note that this may go past the checkpoint entry if need
    //   be to undo writes by transactions that were
    //   active at the time of the checkpoint
    // This is called the undo phase
    if the entry is a <write> and the transaction is on
            the list of transactions needing to be undone
        undo it
    if the entry is a <start> and the transaction is on
            the list of transactions needing to be undone
        remove it from the list
```

PROJECT

IV. **Shadow Paging**

A. Another method that has been used for handling recovery is quite different, 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 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.