Recovery Techniques

- The classical UNIX philosophy:
  - When things go bad, broadcast a warning and reboot.
  - Provide no backup, other than tape archives.

- This strategy is definitely not acceptable for a DBMS.

- There is even debate as to whether it is acceptable for an OS. See the classic book *The UNIX Hater's Handbook* from 1994, which is now available on-line:

`http://research.microsoft.com/~daniel/unix-haters.html`

- In most database-management applications, it is critical that integrity be maintained in the face of failures.

Types of failures:

- **Transaction Failure**:
  - This is a logical failure, caused by a deadlock or other reason that the transaction cannot be completed.
  - Examples:
    - Deadlock
    - Programming error in transaction
  - Recovery uses logs written to primary and/or secondary storage as a resource.

- **System Failure**:
  - This is a temporary failure, which wipes out primary memory, but not secondary (disk memory).
  - Types
    - Software failure
    - Hardware or power failure
  - Recovery uses logs which were written to secondary storage (disks).

- **Media Failure**: This is a failure of secondary storage.
  - Disk failure
  - Recovery depends upon access to a supporting archive, such as tape.

- We will focus largely upon transaction failures, with some attention paid to system failures.

The Recovery Manager:

The heart and brains of the recovery management process is the *recovery manager.*

It handles at least three distinct types of input:

Transaction reads and writes:

- The recovery manager has the responsibility of:
  - *logging* all writes in a secure way, so that recovery may be effected;
  - handling all reads in such a way that the correct image of the database is accessed.
- It may work with a *buffer manager* in this respect.

Transaction terminators:

- The recovery manager must process *abort* commands from transactions, since portions of other transactions may need to be *undone* (rollback) or *redone*.
- The recovery manager must process *commit* commands from transactions, so it knows which writes are permanent and cannot be aborted.

Recover commands:

- The recovery manager handles explicit recovery requests from the system.

Basic Update Strategies:

To understand recovery management, it is best to start with a few “pure” examples, to illustrate the extremes of update strategies.

Update strategies may be placed into two basic categories (although most practical strategies are a combination of these two):

Deferred update:

- All writes within a transaction are recorded in a temporary log.
- During the lifetime of the transaction, these writes are invisible to other transactions.
- When the transaction is completed, these changes are written to the database in one logically indivisible operation (at the *commit* operation).

Immediate update:

- All writes within a transaction are written directly to the database, where they are visible to other transactions.

The choice of strategies affects:

- The type of action needed for recovery.
- The information which is necessary for the *transaction log* to support recovery.

Example: T_1 = r(x) w(x) r(y) w(y)
T_2 = r(y) w(y) r(z) w(z)

Deferred update:

T_1	T_2	Temp Log	Database
			x_0 y_0 z_0
r(x)			x_0 y_0 z_0
w(x)		x_1	x_0 y_0 z_0
	r(y)	x_1	x_0 y_0 z_0
	w(y)	x_1 y_2	x_0 y_0 z_0
r(y) [y_0]		x_1 y_2	x_0 y_0 z_0
w(y)		x_1 y_2 y_1	x_0 y_0 z_0
Commit		y_2	x_1 y_1 z_0
	r(z)	y_2	x_1 y_1 z_0
	w(z)	y_2 z_2	x_1 y_1 z_0
	Commit		x_1 y_2 z_2

Immediate update:

T_1	T_2	Temp Log	Database
			x_0 y_0 z_0
r(x)			x_0 y_0 z_0
w(x)		x_0	x_1 y_0 z_0
	r(y)	x_0	x_1 y_0 z_0
	w(y)	x_0 y_0	x_1 y_2 z_0
r(y) [y_2]		x_0 y_0	x_1 y_2 z_0
w(y)		x_0 y_0 y_2	x_1 y_1 z_0
Commit		x_0 y_0	x_1 y_1 z_0
	r(z)	x_0 y_0	x_1 y_1 z_0
	w(z)	x_0 y_0 z_0	x_1 y_1 z_2
	Commit		x_1 y_1 z_2

The Transaction Log:

To support the recovery process, the recovery manager maintains an extensive *transaction log.*

- The *physical* configuration of the log varies substantially from system to system.

- From a *logical* point of view, each entry in the log must contain the following information.
    - transaction identity
    - time stamp
        - If the transactions are kept in sequence of occurrence, no time stamp is necessary.
    - specific transaction information

Generic examples (time stamp not shown):

Begin(*Transaction*)

Commit(*Transaction*)

Abort(*Transaction*)

Before_Image(*Transaction, Data_object*)
= The value of *Data_object* before it is written by *Transaction.*

After_Image(*Transaction, Data_object*)
= The value of *Data_object* after it is written by *Transaction.*

Example of Log Entries with Pure Deferred Update:

T_1	T_2	Augment Trans. Log	Database
			x_0 y_0 z_0
Begin		Begin(T_1)	
r(x)			
w(x)		After(T_1,x)	
	Begin	Begin(T_2)	
	r(y)		
	w(y)	After(T_2,y)	
r(y) [y_0]			
w(y)		After(T_1,y)	
			x_1 y_1 z_0
Commit		Commit(T_1)	
	r(z)		
	w(z)	After(T_2,z)	
			x_1 y_2 z_2
	Commit	Commit(T_2)	

- The after image is needed to support the commit operation itself.

- The after image is also needed if the transaction is to be re-done as part of a recovery effort.

- With pure deferred update, no before images of data objects are required.

- Reads need not be recorded in the log.

Recovery with Pure Deferred Update:

<u>From an aborted transaction</u>:

- Nothing need be done (except to update the log), since the aborted transaction did not modify the database.

<u>From a system crash</u>:

- For transactions which committed before the crash, do nothing, since their actions are already recorded in the database.
  - If the database must be recovered from the log, “redo” the transaction.
  - Generally, do not actually re-execute the transaction.
    - However, if the transaction was interactive, this may be the only option.
  - Use the after-images in the log to reconstruct the transaction action.
    - Note that only the last (in terms of commit time of the transaction) after image for each data object actually need be restored, so some preprocessing of the log file is warranted.

- For transactions which started but did not commit before the crash, simply re-run each transaction, since no actions were recorded in the database.
- Undo is never required as part of a recovery, since transactions are only written to the permanent database after a commit.

Example of Log Entries with Pure Immediate Update:

T_1	T_2	Trans. Log	Database
			x_0 y_0 z_0
Begin		Begin(T_1)	
r(x)		Read(T_1,x)	
		Before(T_1,x)	
		After(T_1,x)	
w(x)			x_1 y_0 z_0
	Begin	Begin(T_2)	
	r(y)	Read(T_2,y)	
		Before(T_2,y)	
		After(T_2,y)	
	w(y)		x_1 y_2 z_0
r(y) [y_2]		Read(T_1,y)	
		Before(T_1,y)	
		After(T_1,y)	
w(y)			x_1 y_1 z_0
Commit		Commit(T_1)	
	r(z)	Read(T_2,z)	
		Before(T_2,z)	
		After(T_2,z)	
	w(z)		x_1 y_1 z_2
	Commit	Commit(T_2)	

- The before image is needed if the transaction is to be un-done as part of a recovery effort.

- Reads must be logged to support rollback.

- After images are needed to allow redo rather than require re-run for recovery of committed transactions after a system crash.

Recovery with Pure Immediate Update:

From an aborted transaction:

- A *rollback process* (“undo”) must be initiated:
  - For each write which the transaction made, the before image is used to restore the database state to the value that it had before the transaction modified it.
- The rollback must be *cascaded*:
  - If the rolled-back transaction wrote a value which a second transaction read, that second transaction must be rolled back also. And so on…

From a system crash:

- Transactions which started but did not commit before the crash must be treated as aborted transactions.
  - To ensure proper cascading, rollback must proceed in reverse time order.

- For transactions which committed before the crash:
  - Their actions are already recorded in the database.
  - They only need be rolled back if mandated by a cascading relationship.
  - If the database itself is compromised, the after images in the log may be used to redo the transactions.

Recoverable Schedules

Consider the following situation with immediate update:

T_1	T_2	Trans. Log	Database
			x_0 y_0 z_0
Begin		Begin(T_1)	
r(x)		Read(T_1,x)	
		Before(T_1,x)	
		After(T_1,x)	
w(x)			x_1 y_0 z_0
	Begin	Begin(T_2)	
	r(y)	Read(T_2,y)	
		Before(T_2,y)	
		After(T_2,y)	
	w(y)		x_1 y_2 z_0
r(y) [y_2]		Read(T_1,y)	
		Before(T_1,y)	
		After(T_1,y)	
w(y)			x_1 y_1 z_0
Commit		Commit(T_1)	
	Abort		

To recover, transaction T_1, which has committed, must be rolled back. This is very difficult to achieve.

A schedule is called *recoverable* if, for any two transactions T_i and T_j, if T_i writes a data item whose value T_j reads, then T_i must commit before T_j does.

In a recoverable schedule, to effect recovery, committed transactions need never be rolled back.

Unsuitability of the Pure Update Strategies

- Both pure deferred update and pure immediate update are unrealistic to employ in practice.

- Disadvantage of pure deferred update (Redo with no undo):

    - It requires a huge disk cache to maintain all of the updates which have been executed but not committed to the permanent database.

- Disadvantage of pure immediate update:

    - It requires a huge disk cache to maintain all of the before images.

    - The possibility of long, cascaded rollbacks makes it particularly unsuitable in its pure form.

    - Guaranteeing recoverability is a complex task.

    - It is difficult to envision **any** practical database system which would require cascaded rollback as part of its recover strategy.

Basic Properties of Every Recovery Algorithm:

Before looking at less ideal but more effective strategies, it is useful to identify some key points which must be kept in mind, regardless of approach.

Commit point:

- Every transaction has a *commit point.* This is the point at which it is finished, and all of the database modifications which it mandates are made a permanent part of the database.
- Once it has committed, a transaction can no longer be aborted (although it may need to be "undone" as part of a recovery process.)
- Under some strategies, a transaction may modify the database before it has committed.

Every recovery algorithm must meet the following two conditions:

Write-ahead-log protocol: In the case that a transaction may write the database before it commits, the before image of every database object which is modified by a transaction must be written to the log before the after image is written to the database.

Commit rule: The after image of every object written by a transaction must be written to permanent memory (to the log or to the database itself) before that transaction *commits.*

DBMS Caches and Checkpoints

- Any practical logging and recovery strategy will make use of a fast but volatile *DBMS cache* (also called a *DBMS buffer*) to store data temporarily.

- Such caches typically reside in main memory.

- They are part of the DBMS and should not be confused with memory and/or disk caches of the hardware or operating system.

- Such a strategy may also make use of fast but volatile in-memory index structures to temporary or permanent data on disk.

- In the event of a system crash, the data in these volatile memories will be lost.

- At a *checkpoint,* the following steps are taken:

  1. All active transactions are suspended.

  2. All data residing in the DBMS cache and other DBMS memory buffers are written to the corresponding permanent locations on disk.

  3. Copies of volatile index structures are updated on permanent secondary storage.

  4. The checkpoint existence is written to the log.

  5. The suspended transactions are resumed.

Pages in the DBMS Cache

- The DBMS cache is divided into *pages.*

- Such pages typically have two associated bits.

- The *dirty bit* has its usual rôle in a cache:

  - The bit is initially set to 0.
  - It is set to 1 when the cache page has been modified, but not yet written to disk.

- The *pin-unpin bit* has the following rôle:

  - If the page may be written to disk, this bit is set to 0.
  - If the page may not be written back to disk, this bit is set to 1, and the page is said to be *pinned.*

- When is a page pinned?

  - Pinning occurs when a transaction has locked a data object associated with it, so that its current contents is not useful in a more global context.
  - Note that pinned pages cannot be written in place (as permanent archives) at a checkpoint without further control measures.
  - For this reason, a pinned page is often *shadowed* with its most recent unpinned instance, and this shadowed instance is written to disk at the checkpoint.

In-Place vs. Shadowed Updating of the DBMS Cache

In-Place updating:

- In *in-place updating,* a page of the DBMS cache is associated with a single, unique page on the permanent secondary storage.

- A cache flush always writes each page of the disk cache to the associated page in secondary storage.

Shadow updating:

- In *shadow updating*, a page of the DBMS cache may be associated with several (usually two) pages on the permanent secondary storage, all associated with the same logical data item(s). Thus, there may be several permanent copies of a data item.

- A cache flush may write the page of the disk cache to a new location on the secondary storage, thus preserving the original data on that store.

- By keeping *before* and *after* images on disk as shadow updates, the need for such entries in the log is obviated.

- A specific model, using *shadow paging*, will be described shortly.

Some Modified Approaches to Recovery Management

- There are many approaches to recovery management which combine basic principles with tricks of the trade to improve performance.

- These slides will sketch a few:

  - Approaches with *stealing.*
  - Approaches with *no-force* writes of the disk cache.
  - Shadow paging.

- Real strategies combine ideas from all of these, with many others.

Steal vs. No-Steal Approaches

- These strategies relate the point at which a page of the DBMS cache may be written to permanent secondary storage to the point at which a transaction commits.

- Whether or not such pages may be written is governed by the value of the pin/unpin bit.

- In a *no-steal* approach, updates made to data items in pages cannot be written to disk until the transaction has committed.

- This is the default in a deferred update approach.

Modified deferred update with stealing:

- In a *steal* approach, pages may be written to disk before the transaction commits.

- This may reduce greatly the size of DBMS cache needed.

- If another transaction needs to read or write the stolen page, it must wait until the transaction which wrote it previously commits.

- If the transaction which wrote the page is aborted, the previous value of the page must be recovered from the log.

- This strategy is widely used in practice.

Force vs. No-Force Approaches

- These strategies relate the point at which a page of the DBMS cache must be written to permanent secondary storage to the point at which a transaction commits.

- In a *force* approach, at the commit point of a transaction, all pages which were updated by the transaction are written to permanent secondary storage immediately.

- This is the default in a deferred update approach.

<u>Modified deferred update without forcing:</u>

- In a *no-force* approach, pages need not be written to disk immediately after the transaction commits.

- This may reduce greatly the number of disk writes needed, since active pages will remain in memory.

- If the system crashes before the page is written, it must be recovered from the system log.

- This strategy is widely used in practice.

Shadow Paging for Database Systems

- This approach is related to shadow updating of disk caches. However, the emphasis here is upon the directory itself, and not how it is maintained.

- For each transaction, two directories to the database pages for the data objects which the transaction will need are maintained: (see Figure 19.5 of the textbook (21.5 in the 3rd edition).)
  - current directory;
  - shadow directory.

- Initially, these two directories contain identical entries.

- During execution of the transaction, neither the shadow directory nor the pages to which it points are modified.

- Instead, upon first modification of an object, a copy is created, and the current directory set to refer to that copy. Only that copy may be updated by the transaction.

- When the transaction commits, the shadow directory is removed, and the corresponding pages are freed.

- If the transaction aborts, the current directory is removed, and the shadow directory is restored to be the current one.

- Ideally, this is almost a no-undo/no-redo strategy. (To be truly so, directory update would have to be performed in a single uninterruptable step.)

- In practice, things are not so ideal:

  - The pages on which the database is stored become very fragmented. This can lead to greatly increased access times, since adjacency of pages which are likely to be retrieved together cannot be maintained.

  - Garbage collection becomes a significant problem.

A modified strategy:

- Keep all shadowed pages (those associated with the current directory, but not the shadow directory) in the DBMS cache.

- When the transaction commits, write those pages to the permanent database in secondary memory.

- This is then nothing more than a form of caching.

- The main advantage of shadow paging, that disk pages need not be rewritten upon commit, is lost.

If the directory is maintained via the page table, then this approach corresponds to shadow updating.

Log Security:

A fundamental property of the transaction log is that it must be *secure*.

- In the event of a system crash:
  - It <u>must</u> be possible to restore the system to a consistent state, in which it is known exactly which transactions completed and which were aborted.
  - It is important that as little of the completed transactions as possible be lost in the event of a crash.

- The second point leads to a tradeoff decision during the design process:
  - To protect information in the event of a system crash, it is generally necessary to save it to secondary storage.
  - Maintaining the entire log on secondary storage implies a serious performance compromise.

- The solution is generally to maintain some of the log in main memory, but in such a way as to minimize the above problems.
  - Keep the log on a separate disk drive with a separate channel.
    - Speeds up operation
    - Allows recovery even in the case of a crash of the main disk.
  - Write to the log disk as frequently as possible.

Summary of Before and After Images:

Suppose that transaction T issues a write w(x) on data object x.

- The value of x before the write is called the *before image* of x.

- The value of x after the write is called the *after image* of x.

A general recovery manager must maintain both of these images in the log.

- The before image is needed in case the transaction must be “undone.”

- The after image is needed in case the transaction must be “redone.”

- These values need not always be represented explicitly. There are tricks which have been developed to maintain things securely while reducing the amount of physical storage necessary.

Summary of Undo and Redo and Rerun Operations:

To effect a recovery, there are three basic forms of operations which the recovery manager may need to carry out:

- Undo: A transaction needs to be “reversed.” The updates which a transaction has made to the database may need to be “undone,” in order to restore the database to a useful state.

- Redo: The changes to the database effected by a transaction were lost in the failure, so these changes must be recovered from the log and applied to the database.

- Rerun: The changes to the database effected by a transaction were lost in the failure. However, for some reason, they cannot be recovered from the log. Thus, the entire transaction must be run again.

- With pure deferred updates, only redo operations are required.

- With pure immediate updates, only undo operations (with a subsequent rerun) are required to recover from transaction failure.

- Rerun is needed when the transaction did not commit reliably; for example, when it is rolled back after an undo. It is generally implied after an undo.

Recovery strategies are often classified according to the operations which they require:

- Only redo operations required
- Only undo operations required
- Both redo and undo operations required
- Neither undo nor redo operations required

Recovery from Disk Crashes

- Recovery from disk crashes is much more difficult than recovery from transaction failures or machines crashes, because the second line of storage is lost.

- Loss from such crashes is much less common today than it was previously, because of the wide use of redundancy in secondary storage (RAID technology).

- Nonetheless, some protection against such crashes must be built into the system, and this is usually done in several ways.

  - The log for the database system is usually written on a separate physical disk from the database. This log may be used to rebuild the database, at least partially, in the event of a crash.

  - Periodically, the database is also backed up to tape or other archival storage.

Using the Log to Reconstruct the Database:

So far, we have only looked at using the log to recover from transaction failures and system crashes.

It may also be used, to a degree, to recover from disk crashes.

Assumption:

- The transaction log is kept on a separate disk from the database.
- This is a good safety measure in any case.

Requirement:

- The log must record after-images, regardless of the update strategy.

Algorithm:

- Assume that the database has been saved to a further archive at time point t.
- Assume that the log contains after images for all transactions which occurred after time point t.
- The database may then be restored by installing the after images to (a copy of) the archive.
- If immediate update or a more complex strategy is used, further processing may be necessary on this restored copy, as per the previous algorithms.