Recovery Methods

5DV052 — Advanced Data Models and Systems
Umeå University
Department of Computing Science

Stephen J. Hegner
hegner@cs.umu.se
http://www.cs.umu.se/~hegner

Spring 2011

The Issue of Concurrency in the DBMS Context

- In the domain of operating systems, the focus with recovery is to restore the system to a working state as quickly as possible.
- Restoring applications and storage to the states they were in when the failure occurred is not a priority, and is considered the responsibility of the application itself.
- In the domain of database systems, the emphasis is very different.
- The integrity of both the database and of the transactions is of the highest priority.
- For any type of failure, it must always be possible to:
  - Restore the system to a consistent state.
  - Know exactly which actions were committed to the database and which were aborted.

Types of Failures in Database Systems

Transaction failure: A transaction failure can occur in two ways.

1. The transaction itself cannot continue for internal reasons (*e.g.*, aborted by user, necessary input not available, programming error).
2. The transaction must be aborted by the system for some reason (*e.g.*, deadlock).

- In either case, recovery uses logs written to primary and/or secondary storage.

System failure: System failures are those in which primary memory, but not in general secondary memory (*e.g.*, disks), is lost.

- Examples include software failures, hardware failures, and power failures.
- Recovery generally uses logs written to secondary storage.

Medium failure: This is a failure of secondary storage.

- Recovery typically uses alternate secondary storage or tertiary storage (*e.g.*, tape backup).

- The focus in these lectures will be upon transaction failures.

The Recovery Manager

- At the center of the recovery process is the *recovery manager*.
- It handles three distinct types of input.

Transaction reads and writes: The recovery manager has the responsibility:

- to *log* all writes is a secure way;
- to manage reads in such a way that the correct image of the database is accessed.

Transaction terminators: The recovery manager must:

- process *aborts* of transactions, since portions of other transactions may need to be *undone* (rollback) or *redone*;
- process *commits* of transactions, so it is known which writes are permanent and cannot be aborted.

Recover commands: The recovery manager handles explicit recovery requests from the system.

Pure Update Strategies

- To understand recovery management, it is best to start with two "pure" variants, even though most practical strategies involve a combination of these two and other "tricks" as well.

Immediate update: All write operations of a transaction result in immediate updates to the main database, where they are visible to other transactions.

Deferred update: All write operations of a transaction are entered into a log, which is not visible to other transactions.

  - When the transaction commits, the updates in these log entries are entered into *the stable database* (the main database on non-volatile storage) where they become visible to other transactions.

- The choice of strategy affects:
  - the type of action required for recovery, and
  - the information which is necessary for the *transaction log* to support recovery.
- Each of these pure strategies will be next be discussed within SVCC.

Examples of Pure Update Strategies

- Consider:

$$T_1 = \mathrm{r}_1\langle x\rangle \mathrm{w}_1\langle x\rangle \mathrm{r}_1\langle y\rangle \mathrm{w}_1\langle y\rangle$$
$$T_2 = \mathrm{r}_2\langle y\rangle \mathrm{w}_2\langle y\rangle \mathrm{r}_2\langle z\rangle \mathrm{w}_2\langle z\rangle$$

Immediate Update				Deferred Update			
T_1	T_2	TmpLog	DB	T_1	T_2	TmpLog	DB
$\mathrm{r}_1\langle x\rangle$			$x_0 y_0 z_0$	$\mathrm{r}_1\langle x\rangle$			$x_0 y_0 z_0$
$\mathrm{w}_1\langle x\rangle$		x_0	$x_1 y_0 z_0$	$\mathrm{w}_1\langle x\rangle$		x_1	$x_0 y_0 z_0$
	$\mathrm{r}_2\langle y\rangle$	x_0	$x_1 y_0 z_0$		$\mathrm{r}_2\langle y\rangle$	x_1	$x_0 y_0 z_0$
	$\mathrm{w}_2\langle y\rangle$	$x_0 y_0$	$x_1 y_2 z_0$		$\mathrm{w}_2\langle y\rangle$	$x_1 y_2$	$x_0 y_0 z_0$
$\mathrm{r}_1\langle y\rangle[y_2]$		$x_0 y_0$	$x_1 y_2 z_0$	$\mathrm{r}_1\langle y\rangle[y_0]$		$x_1 y_2$	$x_0 y_0 z_0$
$\mathrm{w}_1\langle y\rangle$		$x_0 y_0 y_2$	$x_1 y_1 z_0$	$\mathrm{w}_1\langle y\rangle$		$x_1 y_2 y_1$	$x_0 y_0 z_0$
cmt_1		$x_0 y_0$	$x_1 y_1 z_0$	cmt_1		y_2	$x_1 y_1 z_0$
	$\mathrm{r}_2\langle z\rangle$	$x_0 y_0$	$x_1 y_1 z_0$		$\mathrm{r}_2\langle z\rangle$	y_2	$x_1 y_1 z_0$
	$\mathrm{w}_2\langle z\rangle$	$x_0 y_0 z_0$	$x_1 y_1 z_2$		$\mathrm{w}_2\langle z\rangle$	$y_2 z_2$	$x_1 y_1 z_0$
	cmt_2		$x_1 y_1 z_2$		cmt_2		$x_1 y_2 z_2$

Data item subscripts:

0 $\Rightarrow$ original data; 1 $\Rightarrow$ written by T_1; 2 $\Rightarrow$ written by T_1.

The Transaction Log

- To support the recovery process, the recovery manager maintains an extensive *transaction log*.
- The *physical* configuration of the log varies substantially amongst implementations.
- From a *logical* point of view, each entry in the log file must contain the following information.
  - transaction identity
  - time stamp
  - specific information about the transaction

Form of Entries in the Transaction Log

- Entries in the transaction log might have the following format:
  - For simplicity, time stamps are not shown, but such a stamp is associated with each object.

Begin(`Transaction`) Indicates that `Transaction` has begun.

Commit(`Transaction`) Indicates that `Transaction` has committed.

Abort(`Transaction`) Indicates that `Transaction` has aborted.

Before_Image(`Transaction`,`Data_Object`) The value of `Data_Object` before it was written by `Transaction`.

After_Image(`Transaction`,`Data_Object`) The value of `Data_Object` after it was written by `Transaction`.

Read(`Transaction`,`Data_Object`) Indicates that `Transaction` performed a read on `Data_Object`.

Write(`Transaction`,`Data_Object`) Indicates that `Transaction` performed a write on `Data_Object`.

Example of Log Entries with Pure Immediate Update

Immediate Update			
T_1	T_2	Trans Log	DB
			$x_0 y_0 z_0$
		Begin$\langle T_1 \rangle$	$x_0 y_0 z_0$
$r_1\langle x \rangle$		Read$\langle T_1, x \rangle$	$x_0 y_0 z_0$
		Before$\langle T_1, x \rangle$	$x_0 y_0 z_0$
		After$\langle T_1, x \rangle$	$x_0 y_0 z_0$
$w_1\langle x \rangle$		Write$\langle T_1, x \rangle$	$x_1 y_0 z_0$
		Begin$\langle T_2 \rangle$	$x_1 y_0 z_0$
	$r_2\langle y \rangle$	Read$\langle T_2, y \rangle$	$x_1 y_0 z_0$
		Before$\langle T_2, y \rangle$	$x_1 y_0 z_0$
		After$\langle T_2, y \rangle$	$x_1 y_0 z_0$
	$w_2\langle y \rangle$	Write$\langle T_2, y \rangle$	$x_1 y_2 z_0$
$r_1\langle y \rangle[y_2]$		Read$\langle T_1, y \rangle$	$x_0 y_2 z_0$
		Before$\langle T_1, y \rangle$	$x_1 y_2 z_0$
		After$\langle T_1, y \rangle$	$x_0 y_2 z_0$
$w_1\langle y \rangle$		Write$\langle T_1, y \rangle$	$x_1 y_1 z_0$
cmt_1		Commit$\langle T_1 \rangle$	$x_1 y_1 z_0$
	$r_2\langle z \rangle$	Read$\langle T_2, z \rangle$	$x_1 y_1 z_0$
		Before$\langle T_2, z \rangle$	$x_1 y_1 z_0$
		After$\langle T_2, z \rangle$	$x_1 y_1 z_0$
	$w_2\langle z \rangle$	Write$\langle T_2, z \rangle$	$x_1 y_1 z_2$
	cmt_2	Commit$\langle T_2 \rangle$	$x_1 y_1 z_2$

- The before image is needed if the transaction is to be un-done (rolled back) as part of a recovery effort.
- Reads must be logged to support rollback.
- After images are required to allow re-do (from log entries) rather than re-run (re-execution of the transaction) for recovery of committed transactions after a system crash.

Recovery with Pure Immediate Update

Recovery from an aborted transaction: a *rollback process* must be initiated:

- For each write which the transaction made, the before image is used to restore the database state to that which was valid just before the transaction modified it.
- Cascading of the rollback to other, non-committed transactions may also be necessary.
  - The before images are used to restore the correct values.
- If the schedule is not recoverable, cascading of rollbacks to committed transactions may be necessary.

Recovery from a system crash:

Transactions which did not commit before the crash: are treated as aborted transactions.

Transactions which committed before the crash:

- Their actions are already recorded in the database.
- If the schedule is recoverable, they never need to be rolled back.
- If the database itself is compromised, the after images in the log may be used to re-do the transactions.

Example of Log Entries with Pure Deferred Update

Deferred Update			
T_1	T_2	Trans Log	DB
			$x_0 y_0 z_0$
		Begin$\langle T_1 \rangle$	$x_0 y_0 z_0$
$r_1\langle x \rangle$		Read$\langle T_1, x \rangle$	$x_0 y_0 z_0$
		After$\langle T_1, x \rangle$	$x_0 y_0 z_0$
$w_1\langle x \rangle$		Write$\langle T_1, x \rangle$	$x_0 y_0 z_0$
		Begin$\langle T_2 \rangle$	$x_0 y_0 z_0$
	$r_2\langle y \rangle$	Read$\langle T_2, y \rangle$	$x_0 y_0 z_0$
		After$\langle T_2, y \rangle$	$x_0 y_0 z_0$
	$w_2\langle y \rangle$	Write$\langle T_2, y \rangle$	$x_0 y_0 z_0$
$r_1\langle y \rangle[y_0]$		Read$\langle T_1, y \rangle$	$x_0 y_0 z_0$
		After$\langle T_1, y \rangle$	$x_0 y_0 z_0$
$w_1\langle y \rangle$		Write$\langle T_1, y \rangle$	$x_0 y_0 z_0$
cmt_1		Commit$\langle T_1 \rangle$	$x_1 y_1 z_0$
	$r_2\langle z \rangle$	Read$\langle T_2, z \rangle$	$x_1 y_1 z_0$
		After$\langle T_2, z \rangle$	$x_1 y_1 z_0$
	$w_2\langle z \rangle$	Write$\langle T_2, z \rangle$	$x_1 y_1 z_2$
	cmt_2	Commit$\langle T_2 \rangle$	$x_1 y_2 z_2$

- 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.
- No before images are required.
- Read operations need not be recored in the log.

Recovery with Pure Deferred Update

Recovery from an aborted transaction: Nothing needs to be done (except to update the log) — the aborted transaction did not modify the database.

Recovery from a system crash:

Transactions which did not commit before the crash: are re-run, since the aborted transactions did not update the database.

- Un-do (rollback) is never required as part of the recovery, since uncommitted transactions never write the database.

Transactions which committed before the crash: have their actions already recorded in the database, so no recovery action is necessary.

If the database must be recovered from the log: re-do the transaction from log entries.

- There is no need to re-execute (re-run) the transaction.
- The updates of the original transaction may be recovered from the after images in the log.
- The last after image (in temporal order) is used as the value for that object in the recovered database.

Basic Properties of Every Recovery Algorithm

- Key points which must be kept in mind, regardless of approach.

Commit point: Every transaction has a *commit point*.

  - It is the point at which it is finished, and the result of its write operations become permanent in the database.
  - Once a transaction has committed, it can no longer be aborted.
  - If a transaction modifies the database before commit, the system must be prepared to undo those modifications in case the transaction does not complete.

- 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 (*i.e.*, to the log or to the database itself) <u>before</u> the transaction commits.

Suitability of the Pure Update Strategies

- Deferred update might seem to be an ideal solution, but it has several practical limitations.

  Performance of a log-centric strategy: The primary issue is that to execute transaction updates via the log would be far too slow.

  - The log is designed primarily for reliability, not speed.
  - Furthermore, since the log can become very large, entries are maintained in a compact format.
  - From a performance point of view, it is not feasible to execute database operations, particularly commit operations, via the log alone.

- But immediate update has its problems as well.

  Too many small writes:

  - In immediate update, each write operation by a transaction requires a write to the database.
  - Large databases are typically held in secondary storage.
  - Thus, a serious and often unacceptable performance hit arises.

The Database Cache

- The solution to the shortcomings of the pure update strategies is to use a *database buffer* or *database cache*.
- The database cache bears the same relationship to the database that a hardware cache does to memory in a computer system.
  - It provides fast, temporary access to frequently needed data items.
  - It employs typical replacement strategies such as LRU.
  - It is typically kept in main (and usually volatile) memory.
  - There is usually no special hardware for the DB cache.
- Proper management of the DB cache is central to its utility.
- The actual strategy supported in SVCC is typically immediate update, but with updates to the cache, not the main database.
- However, the main ideas which will be presented will work with deferred update also.
- A few of the most important management techniques will be discussed next.

Pages in the Database Cache

- The database cache divided into *pages*.
- Each page corresponds to a physical page of the database.
- There are two bits associated with each page.

Dirty bit: This bit has its usual meaning for a cache.

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

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 to disk, this bit is set to 1, and the page is said to be *pinned*.

Question: When is a page pinned?

- Pinning occurs when a transaction has locked a data object associated with that page, so that its current contents is not useful in a more global context.

Flexibility in Writing Cache Pages to the Permanent DB

- One way to increase the performance of a DB system is to limit the number of writes between the DB cache and the stable DB.
- This is typically accomplished by waiting until there is a substantial number of such writes to execute and then *batching them* — doing them all at once.
  - A single large transfer is much faster than many smaller transfers.
- There are two main approaches along these lines.

Force vs. no-force:

- In a *force approach*, a cache page containing committed data must be written to the stable database as soon as the commit occurs.
- In a *no-force* approach, committed data may remain in the cache and be written to the stable database later.

Steal vs. no-steal:

- In a *no-steal approach*, a cache page which whose contents has not yet been committed must not be written to the stable database.
- In a *steal* approach, a cache page which has not yet been committed may nevertheless be written to the stable database.

Force vs. No-Force

Force: All write operations by a transaction which are held in the DB cache must be transferred to the stable database at the time of commit.

No-force: Committed writes are allowed to reside in the cache only.

- In the case of a system crash which also destroys the cache, these writes must be recovered from the system log.
- Observe that the usual protocols for cache management must be followed.
- All references to the database are routed through such a manager.
- If a committed data item is found in the cache, that value must be used, because the value in the stable database may not be valid.
- In general, non-committed data items may not be read by other transactions.

Steal vs. No-Steal

No-steal: Only cache pages corresponding to committed data may be written to the stable database.

Steal: Cache pages which are not yet committed (but are expected to commit soon) may be written to the stable database.

- If the transaction associated with the new data value is aborted, the previous value of the page must be recovered from the log and restored to the database.
  - Log-based recovery is necessary in the case of a system crash as well.
- In general, any access by another transaction to an uncommitted data item which has written to the stable database must be blocked until the writer commits.
- However, in the case of RU (read-uncommitted) isolation, reads of uncommitted cache entries may be allowed.
  - This illustrates why RU was conceived.
  - Access to such uncommitted items in the cache is faster than retrieving the true values from the stable database.

Checkpoints

- While recovery from a system crash using the logs alone is possible, it can be a very slow process.
- To make crash recovery more feasible, *checkpoints* are widely used.
- Roughly speaking, at a *checkpoint*, the cache is flushed completely to the stable database, and copies of other volatile items are made.
- In the case of a crash, the database may be restored to its state at the last checkpoint, and the recovery process may commence from that point.

Basic checkpointing: The following five steps are taken:

1. All active transactions are suspended, and no new transactions are allowed to begin.
2. The cache is scanned, and all dirty pages which are not pinned are written to the stable database.
3. Volatile index structures are copied to permanent storage.
4. The existence of the checkpoint is written to the log.
5. Normal operations are allowed to resume, including the remaining actions of suspended transactions.

Fuzzy Checkpointing

- A drawback of basic checkpointing is that all transactions must be suspended during the entire checkpoint process.
- This can be a serious performance issue, particularly real-time and interactive systems.
- For that reason, a more complex variant known as *fuzzy checkpointing* is often used.
- The steps of this process are given on the next slide.

Fuzzy Checkpointing — 2

Fuzzy checkpointing: The following steps are taken.

1. All active transactions are suspended, and no new transactions are allowed to begin.
2. The cache, is scanned, and a list of all dirty pages and all pinned pages is made.
3. Volatile index structures are copied to permanent storage.
4. A list of all active transactions, as well as a pointer to the latest log entry of each, is made.
5. A checkpoint record, including the list created in the previous step, is written in the log.
6. Normal operations are allowed to resume, including the remaining actions of suspended transactions.
7. In parallel with normal operations, operations to flush all dirty pages in the cache to the stable database are made. The latter operations are of lower priority.

- A new fuzzy-checkpoint operation is not allowed to begin until the final step of the previous fuzzy checkpoint has completed.

Optimization of Logging

- The efficiency of logging operation has a profound effect upon the performance of a DBMS.
- Consequently, there has been much work on the problem of making such operations as efficient as possible.
- A few of the most important ideas will be described here.
- They are part of a comprehensive procedure known as *ARIES*.

Granularity of Logging

- On a physical level, the database is stored in *pages*.
- Many logical records may reside on a single page.
- Using physical pages as the basis for the before and after images in log entries is very inefficient.
- It may also lead to problems with aborted transactions.

Example: Consider the following schedule fragment:

$$\mathrm{w}_1\langle x\rangle \mathrm{w}_2\langle y\rangle \mathrm{abort}_1 \mathrm{cmt}_2$$

- Suppose that x and y reside on the same page.
- If the before image of the entire page is restored upon the abort of T_1, the update of T_2 on y will be lost as well.
- The solution is to have the before and after images contain only information on the records which were changed.
- Even better, for large records, only descriptors of the changes to those records need be stored.

Log Sequence Numbers

- To employ record-level images complicates the restart algorithm which is used after a crash.
- It becomes necessary to know whether a given before or after image should be applied to the corresponding page.

Log sequence numbers: Each log entry is assigned a *log sequence number (LSN)*, with later entries having larger LSNs.

- Each page has a header which contains the LSN of the last log operation which identifies an update to that page.
- After a crash, an update operation is re-performed during the restart operation only if the LSN of the log entry is larger than the LSN on the associated physical page.

Problems with Logging Aborts

Question: How are LSNs associated with aborts?

- Using the LSN of the last log record before those which must be reversed might wipe out valid operations.
- In the example below, assume that x and y reside on the same page.

Log entries

LSN = 110	LSN = 111	LSN = 112	LSN = 113
$w_1\langle x\rangle$	$w_2\langle y\rangle$	cmt_2	$abort_1$

Images of the page containing x and y

LSN = 100	LSN = 110	LSN = 111	LSN =?
⋮	⋮	⋮	⋮
x_0	x_1	x_1	x_0
y_0	y_0	y_2	y_2

- It is not clear which LSN should be associated with the abort.
- Using 110 will result in the loss of the committed update by T_2.
- The solution is to log an abort as one or more *undo* operations.

Compensation Records in the Log

- In a *compensation log record*, an operation which is aborted is undone.

LSN = 110	LSN = 111	LSN = 112	LSN = 113	LSN = 113
$w_1\langle x\rangle$	$w_2\langle y\rangle$	cmt_2	$Undo\langle w_1\langle x\rangle\rangle$	$abort_1$

LSN = 100	LSN = 110	LSN = 111	LSN = 113
⋮	⋮	⋮	⋮
x_0	x_1	x_1	x_0
y_0	y_0	y_2	y_2

- The LSN associated with the undo is used for the record image after the abort.
- The log entry for the abort itself is not used in the header of any page.
- With this approach, committed and aborted transactions take essentially the same form and require the same form of recovery.
- The abort entry in the log then is regarded as a commit of a transaction with no net effect.

Managing Undo During Restart

- During a system restart after a crash, the recovery manager must roll back all active transactions by undoing their updates.
- If an active transaction T_i had already aborted when the crash occurred, but the abort-management process had not completed, the recovery manager will see T_i as an active transaction which must be rolled back.
- This will result undoing the undo operations, which is redundant.
- To avoid this problem, each undo can be linked to the log record for the operation which it undoes.
- When an undo record is encountered in the recovery process, it is skipped.
- Instead, the operation of the record to which it points is undone.

Logging Cache Flushes

- It is also useful to log cache flushes with *flush records*.
- During a restart, that information is useful in indicating which updates are already reflected in the stable database and which were only in the cache and must be restored.

Dirty-page table: During the fuzzy-checkpointing operation, each dirty page which is identified is augmented with the lowest-numbered LSN which must be redone to yield that page clean.

- This table has a clear use in speeding up the recovery process.

Log Security

- A fundamental property of the transaction log is that it must be secure.
- In the event of a system crash:
  - It must 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 few of the completed transactions as possible be lost in the even of a crash.
- This last point leads to a tradeoff decision during the design process.
  - To protect data in the event of a system crash, it is necessary to save it to non-volatile storage, which typically means (slow) secondary storage.
  - However, maintaining the entire log on secondary storage would entail a serious performance penalty.
- Using high-speed data links, the log may be replicated on several machines, each with power backup, so that loss of one machine does not compromise the log.

Recovery from Failure of the Stable Database

- Recovery from disk crashes is much more difficult than recovery from transaction failures or machine crashes, because the second line of storage is lost.
- Loss from such crashes is far less common today than it was previously, for at least two reasons:

  Storage redundancy: Modern RAID technology protects against the failure of single drives.

  Redundancy by distribution: Modern, high-speed networks permit databases to be replicated at distinct sites, allowing protections even from events such as fires and terrorist attacks.
- It is nevertheless necessary to build such protections into the system.
- In addition to the above points, the following are central.
  - The DBMS log is typically written to a separate physical disk from the database itself. This "disk" is usually a highly redundant RAID.
  - Automated tertiary backup (to archival tape) is also still a reasonable option.

Recovery under MVCC

- Recovery techniques under MVCC are similar to those for SVCC.
- However, the log need not contain as much information.
- Since they are contained in the various versions of the data items, data values for before and after images need not be maintained in the log.
    - Only references to those values need be maintained.
- This solution assumes that the stable database is as reliable as the log.
    - If not, the risk of loss increases.
    - It is usually necessary to log in full updates which are held only in the cache and not yet in the stable database.
    - This may be accomplished by having a separate log area for temporary data values.
    - These entries would have the same form as usual log entries for SVCC.
    - The log entries can then point to these temporary values until they are entered into the stable database.

Read-Uncommitted Isolation under MVCC

- It was previously noted that RU isolation is not a natural fit to version-based MVCC.
- However, there is one way in which it might make sense.
- Rather than using values in the stable database, an implementation of RU isolation could use data values in the cache, even on pinned pages.
- This could lead to an increase in performance, because those data values would not need to be fetched from the stable database.
- The utility of this approach is dependent in some degree upon how transactions are allowed to use pinned data items.
- If they may hold any "garbage" while the transaction is running, and not just values waiting to be committed, then this approach is questionable.
- It is not clear that it is used in practice.