**UNIT V-Recovery System**

**Topic:Failure Classification**

* **Transaction failure** :
  + **Logical errors**: transaction cannot complete due to some internal error condition like bad i/p,overflow,data not found
  + **System errors**: the database system must terminate an active transaction due to an error condition (e.g., deadlock)
* **System crash**: a power failure or other hardware or software failure causes the system to crash.
  + **Fail-stop assumption**: non-volatile storage contents are assumed to not be corrupted by system crash
    - Database systems have numerous integrity checks to prevent corruption of disk data
* **Disk failure**: a head crash or similar disk failure destroys all or part of disk storage
  + Destruction is assumed to be detectable: disk drives use checksums to detect failures

**Topic: Recovery Algorithms:**

* Recovery algorithms are techniques to ensure database consistency and transaction atomicity and durability despite failures
* Recovery algorithms have two parts
  1. Actions taken during normal transaction processing to ensure enough information exists to recover from failures
  2. Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability

**Topic: Recovery and Atomicity:**

* Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state.
* Consider transaction *Ti* that transfers $50 from account *A* to account *B*; goal is either to perform all database modifications made by *Ti*or none at all.
* Several output operations may be required for *Ti* (to output *A* and *B*). A failure may occur after one of these modifications have been made but before all of them are made.
* To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.
* We have two approaches:
  + **log-based recovery**, and
  + **shadow-paging**

We assume (initially) that transactions run serially, that is, one after the other

1. **Log-Based Recovery**

* A **log** is used for recording database modifications which is kept on stable storage.
  + The log is a sequence of **log records**, and maintains a record of update activities on the database.
* There are several log records
  + Update log record describes a single database write. It has

these fields:

* + - * + Transaction identifier, which is the unique identifier of the transaction that performed the write operation
        + Data-item identifier, which is the unique identifier of the data item written
        + Old value, which is the value of the data item prior to the write
        + New value, which is the value that the data item will have after the write
* When transaction *Ti*starts, it registers itself by writing a   
  *<Ti* start>log record
* *Before Ti*executes write(*X*), a log record *<Ti, X, V1, V2>*is written, where *V1* is the value of *X* before the write, and *V2*is the value to be written to *X*.
  + Log record notes that *Ti* has performed a write on data item *Xj  Xj*had value *V1*before the write, and will have value *V2*after the write.
* When *Ti* finishes it last statement, the log record <*Ti*commit> is written.
* When Ti has failed the log record<Ti, abort> is written.
* Every log record is written to the end of the log on stable storage as soon as created
* Two approaches using logs
  + Deferred database modification
  + Immediate database modification

**Database modification**

* A transaction has to follow the steps to modify a data item:
  + The transaction performs some computations in its own private part of main memory
  + The transaction modifies the data block in the disk buffer in main memory
  + The database system executes the output operation that writes the data block to disk
* Transaction modifies the database, performs an update on a disk buffer or on disk. Updates to the private part of main memory do not count as database modifications.
* There are two approaches using logs
  + Deferred database modification
  + Immediate database modification

**Deferred Database Modification**

* The **deferred database modification** scheme records all modifications to the log, but defers all the **write**s to after partial commit.
* Assume that transactions execute serially
* Transaction starts by writing *<Ti****start****>*record to log.
* A **write**(*X*) operation results in a log record *<Ti, X, V>*being written, where *V* is the new value for *X*
  + Note: old value is not needed for this scheme
* The write is not performed on *X* at this time, but is deferred.
* When *Ti*partially commits, <*Ti***commit**> is written to the log
* Finally, the log records are read and used to actually execute the previously deferred writes.
* During recovery after a crash, a transaction needs to be redone if and only if both *<Ti***start**> and<*Ti* **commit**> are there in the log.
* Redoing a transaction *Ti*( **redo***Ti*) sets the value of all data items updated by the transaction to the new values.
* Crashes can occur while
  + the transaction is executing the original updates, or
  + while recovery action is being taken
* example transactions *T0*and *T1*(*T0*executes before *T1*):

*T0*: **read** (*A*) *T1*: **read** (*C*)

*A: - A - 50* *C:- C- 100*

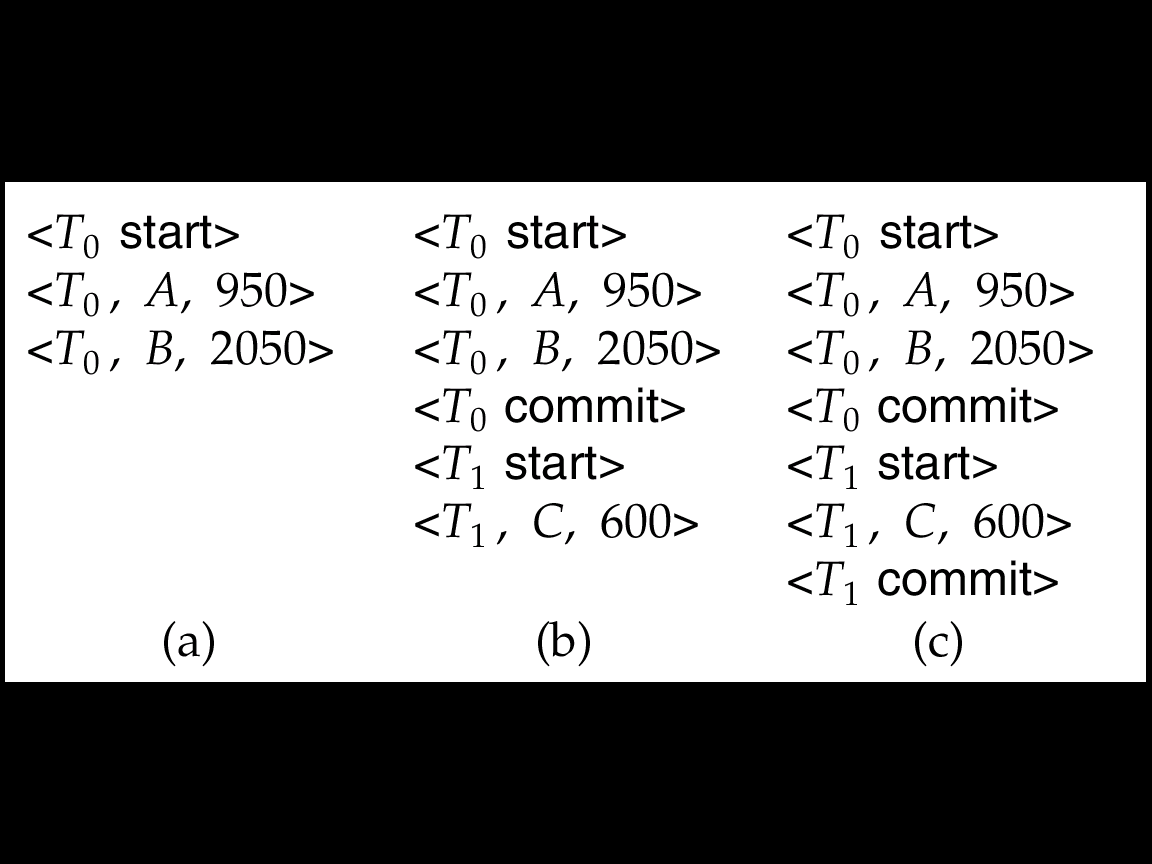
**Write** (*A*) **write** (*C*)

**read** (*B*)

*B:- B + 50*

**write** (*B*)

* Below we show the log as it appears at three instances of time.



* If log on stable storage at time of crash is as in case:

(a) No redo actions need to be taken

(b) redo(*T*0) must be performed since <*T*0 **commi**t> is present

(c) **redo**(*T*0) must be performed followed by redo(*T*1) since

<*T*0**commit**> and <*Ti* commit> are present

**Immediate Database Modification**

* The **immediate database modification** scheme allows database updates of an uncommitted transaction to be made as the writes are issued
  + since undoing may be needed, update logs must have both old value and new value
* Update log record must be written *before* database item is written
  + We assume that the log record is output directly to stable storage
  + Can be extended to postpone log record output, so long as prior to execution of an **output**(*B*) operation for a data block B, all log records corresponding to items *B* must be flushed to stable storage
* Output of updated blocks can take place at any time before or after transaction commit
* Order in which blocks are output can be different from the order in which they are written.

**Immediate Database Modification Example**

**Log Write Output**

**<*T*0start>**

**<*T0,* A, 1000, 950>**

***<T*o*,* B, 2000, 2050>**

***A* = 950**

***B* = 2050**

**<*T*0commit>**

**<*T*1start>**

**<*T*1, C, 700, 600>**

***C* = 600**

***BB*, *BC***

**<*T*1commit>**

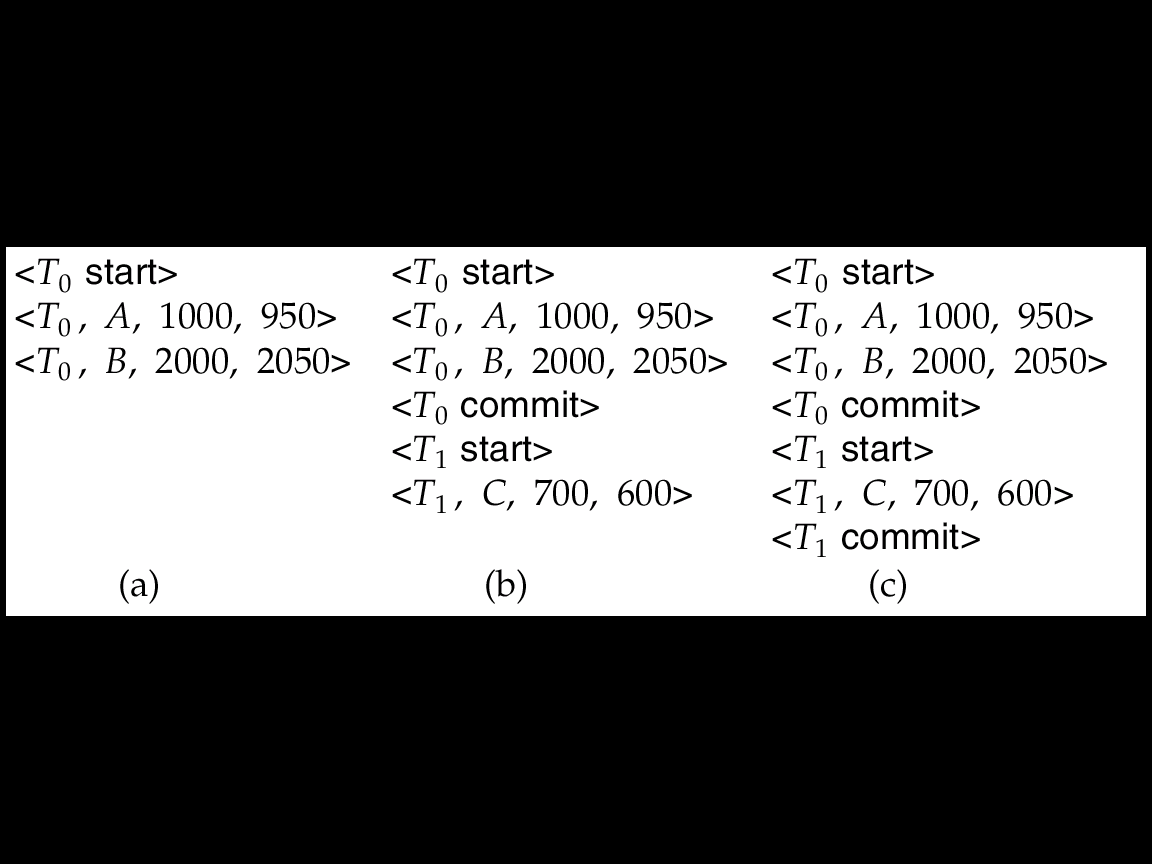
***BA***

Note: *BX*denotes block containing *X*.

* Recovery procedure has two operations instead of one:
  + **undo**(*T*i) restores the value of all data items updated by *Ti* to their old values, going backwards from the last log record for *Ti*
  + **redo**(*T*i) sets the value of all data items updated by *Ti*to the new values, going forward from the first log record for *Ti*
* Both operations must be **idempotent**
  + That is, even if the operation is executed multiple times the effect is the same as if it is executed once
    - Needed since operations may get re-executed during recovery
* When recovering after failure:
  + Transaction *Ti*needs to be undone if the log contains the record   
    *<Ti***start***>*, but does not contain the record *<Ti***commit***>*.
  + Transaction *Ti*needs to be redone if the log contains both the record *<Ti***start***>* and the record *<Ti* **commit***>*.
* Undo operations are performed first, then redo operations.

**Immediate DB Modification Recovery Example**

Below we show the log as it appears at three instances of time.



Recovery actions in each case above are:

(a) undo (*T*0): B is restored to 2000 and A to 1000.

(b) undo (*T*1) and redo (*T*0): C is restored to 700, and then *A* and *B* are

set to 950 and 2050 respectively.

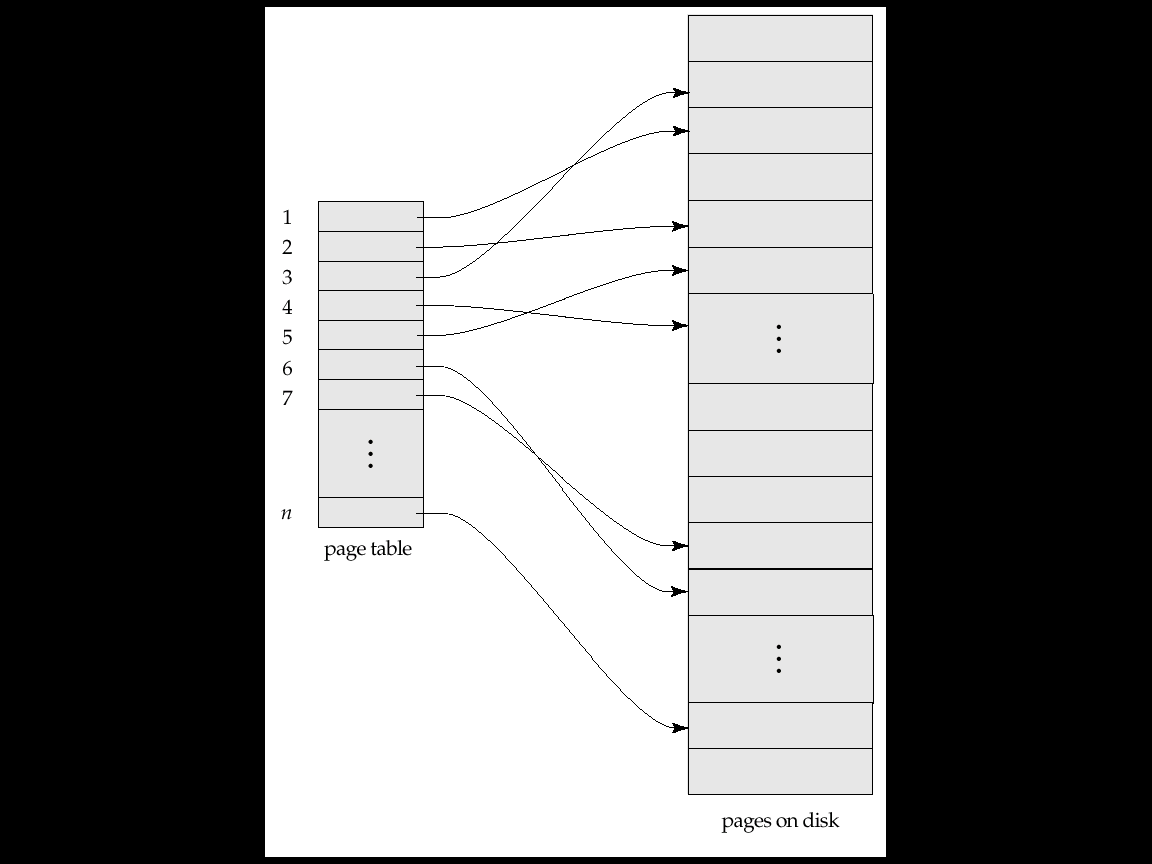
(c) redo (*T*0) and redo (*T*1): A and B are set to 950 and 2050

respectively

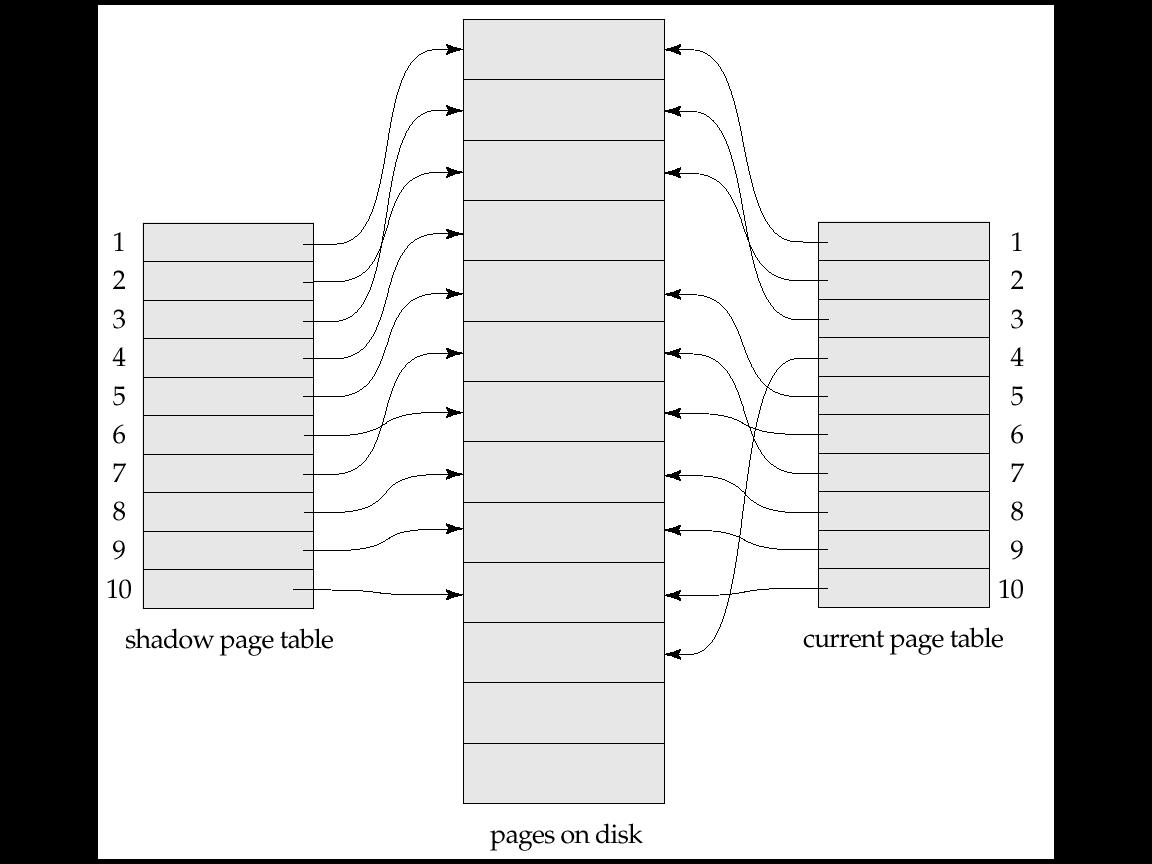
**2) Shadow Paging**

* **Shadow paging** is an alternative to log-based recovery; this scheme is useful if transactions execute serially
* Idea: maintain *two* page tables during the lifetime of a transaction –the **current page table**, and the **shadow page table**
* Store the shadow page table in nonvolatile storage, such that state of the database prior to transaction execution may be recovered.
  + Shadow page table is never modified during execution
* To start with, both the page tables are identical. Only current page table is used for data item accesses during execution of the transaction.
* Whenever any page is about to be written for the first time
  + A copy of this page is made onto an unused page.
  + The current page table is then made to point to the copy
  + The update is performed on the copy

Sample Page Table



Example of Shadow Paging: Shadow and current page tables after write to page 4



* To commit a transaction :

1. Flush all modified pages in main memory to disk

2. Output current page table to disk

3. Make the current page table the new shadow page table, as follows:

* + keep a pointer to the shadow page table at a fixed (known) location on disk.
  + to make the current page table the new shadow page table, simply update the pointer to point to current page table on disk
* Once pointer to shadow page table has been written, transaction is committed.
* No recovery is needed after a crash — new transactions can start right away, using the shadow page table.
* Pages not pointed to from current/shadow page table should be freed (garbage collected).
* Advantages of shadow-paging over log-based schemes
  + no overhead of writing log records
  + recovery is trivial
* Disadvantages :
  + Copying the entire page table is very expensive
    - Can be reduced by using a page table structured like a B+-tree
      * No need to copy entire tree, only need to copy paths in the tree that lead to updated leaf nodes
  + Commit overhead is high even with above extension
    - Need to flush every updated page, and page table
  + Data gets fragmented (related pages get separated on disk)
  + After every transaction completion, the database pages containing old versions of modified data need to be garbage collected
  + Hard to extend algorithm to allow transactions to run concurrently
    - Easier to extend log based schemes

**topic: Database Buffering**

* Database maintains an in-memory buffer of data blocks
  + When a new block is needed, if buffer is full an existing block needs to be removed from buffer
  + If the block chosen for removal has been updated, it must be output to disk
* If a block with uncommitted updates is output to disk, log records with undo information for the updates are output to the log on stable storage first
  + (Write ahead logging)
* No updates should be in progress on a block when it is output to disk. Can be ensured as follows.
  + Before writing a data item, transaction acquires exclusive lock on block containing the data item
  + Lock can be released once the write is completed.
    - Such locks held for short duration are called **latches**.
  + Before a block is output to disk, the system acquires an exclusive latch on the block
    - Ensures no update can be in progress on the block

**Topic: Buffer Management**

* Database buffer can be implemented either
  + in an area of real main-memory reserved for the database, or
  + in virtual memory
* Implementing buffer in reserved main-memory has drawbacks:
  + Memory is partitioned before-hand between database buffer and applications, limiting flexibility.
  + Needs may change, and although operating system knows best how memory should be divided up at any time, it cannot change the partitioning of memory.
* Database buffers are generally implemented in virtual memory in spite of some drawbacks:
  + When operating system needs to evict a page that has been modified, the page is written to swap space on disk.
  + When database decides to write buffer page to disk, buffer page may be in swap space, and may have to be read from swap space on disk and output to the database on disk, resulting in extra I/O!
    - Known as **dual paging** problem.
  + Ideally when OS needs to evict a page from the buffer, it should pass control to database, which in turn should
    - Output the page to database instead of to swap space (making sure to output log records first), if it is modified
    - Release the page from the buffer, for the OS to use

Dual paging can thus be avoided, but common operating systems do not support such functionality.

**Topic: Failure with Loss of Nonvolatile Storage**

* So far we assumed no loss of non-volatile storage
* Technique similar to checkpointing used to deal with loss of non-volatile storage
  + Periodically **dump** the entire content of the database to stable storage
  + No transaction may be active during the dump procedure; a procedure similar to checkpointing must take place
    - Output all log records currently residing in main memory onto stable storage.
    - Output all buffer blocks onto the disk.
    - Copy the contents of the database to stable storage.
    - Output a record <**dump**> to log on stable storage.

**Recovering from Failure of Non-Volatile Storage**

* To recover from disk failure
  + restore database from most recent dump.
  + Consult the log and redo all transactions that committed after the dump
* Can be extended to allow transactions to be active during dump;   
  known as **fuzzy dump** or **online dump**

**Fuzzy Checkpointing**

* **Checkpointing** is done as follows:
  1. Output all log records in memory to stable storage
  2. Output to disk all modified buffer blocks
  3. Output to log on stable storage a < **checkpoint** *L*> record.

Transactions are not allowed to perform any actions while checkpointing is in progress.

* Fuzzy checkpointing allows transactions to progress while the most time consuming parts of checkpointing are in progress

**Topic: ARIES:**

* **ARIES is a state of the art recovery method** 
  + **Incorporates numerous optimizations to reduce overheads during normal processing and to speed up recovery**
  + **The “advanced recovery algorithm” is modeled after ARIES, but greatly simplified by removing optimizations**
* **Unlike the recovery algorithm, ARIES** 
  + **Uses log sequence number (LSN) to identify log records**
    - **Stores LSNs in pages to identify what updates have already been applied to a database page**
  + **Physiological redo**
  + **Dirty page table to avoid unnecessary redos during recovery**
  + **Fuzzy checkpointing that only records information about dirty pages, and does not require dirty pages to be written out at checkpoint time**
* **Physiological redo**
  + **Affected page is physically identified, action within page can be logical**
    - **Used to reduce logging overheads**
      * **e.g. when a record is deleted and all other records have to be moved to fill hole**
        + **Physiological redo can log just the record deletion**
        + **Physical redo would require logging of old and new values for much of the page**
    - **Requires page to be output to disk atomically**
      * **Easy to achieve with hardware RAID, also supported by some disk systems**
      * **Incomplete page output can be detected by checksum techniques,** 
        + **But extra actions are required for recovery**
        + **Treated as a media failure**

**ARIES Data Structures:**

* **ARIES uses several data structures**
  + **Log sequence number (LSN) identifies each log record**
    - **Must be sequentially increasing**
    - **Typically an offset from beginning of log file to allow fast access**
      * **Easily extended to handle multiple log files**
  + **Page LSN**
  + **Log records of several different types**
  + **Dirty page table**

**ARIES Data Structures: Page LSN**

**Each page contains a PageLSN which is the LSN of the last log record whose effects are reflected on the page**

* + **To update a page:**
    - **X-latch the page, and write the log record**
    - **Update the page**
    - **Record the LSN of the log record in PageLSN**
    - **Unlock page**
  + **To flush page to disk, must first S-latch page**
    - **Thus page state on disk is operation consistent**
      * **Required to support physiological redo**
  + **PageLSN is used during recovery to prevent repeated redo** 
    - **Thus ensuring idempotence**

**ARIES Data Structures: Log Record**

* **Each log record contains LSN of previous log record of the same transaction**
  + **LSN in log record may be implicit**
* **Special redo-only log record called compensation log record (CLR) used to log actions taken during recovery that never need to be undone**
  + **Serves the role of operation-abort log records used in advanced recovery algorithm**
  + **Has a field UndoNextLSN to note next (earlier) record to be undone**
    - **Records in between would have already been undone**
    - **Required to avoid repeated undo of already undone actions**

**ARIES Data Structures: DirtyPage Table**

* **DirtyPageTable**
  + **List of pages in the buffer that have been updated**
  + **Contains, for each such page**
    - **PageLSN of the page**
    - **RecLSN is an LSN such that log records before this LSN have already been applied to the page version on disk**
      * **Set to current end of log when a page is inserted into dirty page table (just before being updated)**
      * **Recorded in checkpoints, helps to minimize redo work**

**ARIES Data Structures: Checkpoint Log**

* **Checkpoint log record**
  + **Contains:** 
    - **DirtyPageTable and list of active transactions**
    - **For each active transaction, LastLSN, the LSN of the last log record written by the transaction**
  + **Fixed position on disk notes LSN of last completed  
    checkpoint log record**
* **Dirty pages are not written out at checkpoint time**
  + - **Instead, they are flushed out continuously, in the background**
* **Checkpoint is thus very low overhead**
  + **can be done frequently**

**ARIES Recovery Algorithm**

**ARIES recovery involves three passes**

* **Analysis pass: Determines**
  + **Which transactions to undo**
  + **Which pages were dirty (disk version not up to date) at time of crash**
  + **RedoLSN: LSN from which redo should start**
* **Redo pass:**
  + **Repeats history, redoing all actions from RedoLSN** 
    - **RecLSN and PageLSNs are used to avoid redoing actions already reflected on page**
* **Undo pass:**
  + **Rolls back all incomplete transactions**
    - **Transactions whose abort was complete earlier are not undone**
      * **Key idea: no need to undo these transactions: earlier undo actions were logged, and are redone as required**

**Aries Recovery: 3 Passes**

* **Analysis, redo and undo passes**
* **Analysis determines where redo should start**

**Undo has to go back till start of earliest incomplete transaction**

**ARIES Recovery: Analysis**

**Analysis pass**

* **Starts from last complete checkpoint log record**
  + **Reads DirtyPageTable from log record**
  + **Sets RedoLSN = min of RecLSNs of all pages in DirtyPageTable**
    - **In case no pages are dirty, RedoLSN = checkpoint record’s LSN**
  + **Sets undo-list = list of transactions in checkpoint log record**
  + **Reads LSN of last log record for each transaction in undo-list from checkpoint log record**
* **Scans forward from checkpoint**
* **Scans forward from checkpoint**
  + **If any log record found for transaction not in undo-list, adds transaction to undo-list**
  + **Whenever an update log record is found**
    - **If page is not in DirtyPageTable, it is added with RecLSN set to LSN of the update log record**
  + **If transaction end log record found, delete transaction from undo-list**
  + **Keeps track of last log record for each transaction in undo-list**
    - **May be needed for later undo**
* **At end of analysis pass:**
  + **RedoLSN determines where to start redo pass**
  + **RecLSN for each page in DirtyPageTable used to minimize redo work**
  + **All transactions in undo-list need to be rolled back**

**ARIES Redo Pass**

**Redo Pass: Repeats history by replaying every action not already reflected in the page on disk, as follows:**

* **Scans forward from RedoLSN. Whenever an update log record is found:**
  1. **If the page is not in DirtyPageTable or the LSN of the log record is less than the RecLSN of the page in DirtyPageTable, then skip the log record**
  2. **Otherwise fetch the page from disk. If the PageLSN of the page fetched from disk is less than the LSN of the log record, redo the log record**

**NOTE: if either test is negative the effects of the log record have already appeared on the page. First test avoids even fetching the page from disk!**

**ARIES Undo Actions**

* **When an undo is performed for an update log record**
  + **Generate a CLR containing the undo action performed (actions performed during undo are logged physicaly or physiologically).** 
    - **CLR for record *n* noted as *n*’ in figure below**
  + **Set UndoNextLSN of the CLR to the PrevLSN value of the update log record**
    - **Arrows indicate UndoNextLSN value**
* **ARIES supports partial rollback**
  + **Used e.g. to handle deadlocks by rolling back just enough to release reqd. locks**
  + **Figure indicates forward actions after partial rollbacks** 
    - **records 3 and 4 initially, later 5 and 6, then full rollback**

**ARIES: Undo Pass**

**Undo pass:**

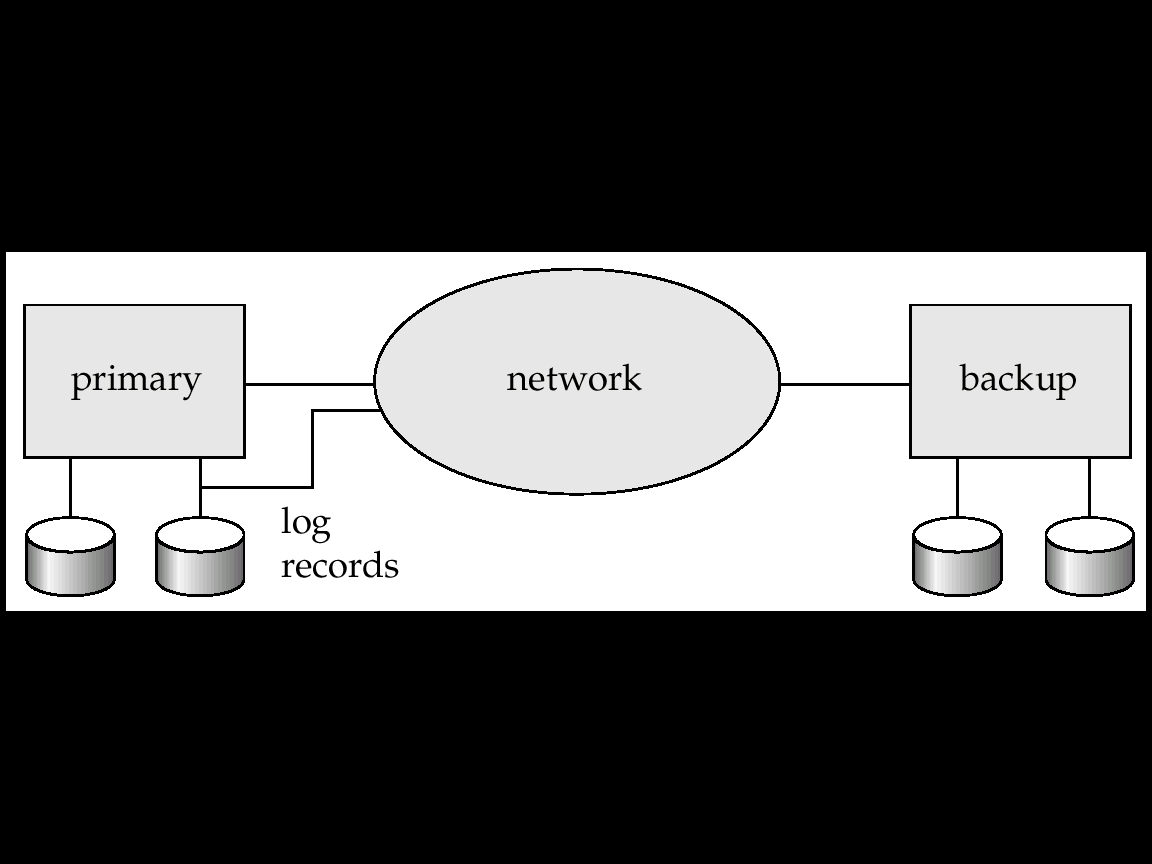
* **Performs backward scan on log undoing all transaction in undo-list**
  + **Backward scan optimized by skipping unneeded log records as follows:**
    - **Next LSN to be undone for each transaction set to LSN of last log record for transaction found by analysis pass.**
    - **At each step pick largest of these LSNs to undo, skip back to it and undo it**
    - **After undoing a log record**
      * **For ordinary log records, set next LSN to be undone for transaction to PrevLSN noted in the log record**
      * **For compensation log records (CLRs) set next LSN to be undo to UndoNextLSN noted in the log record**
        + **All intervening records are skipped since they would have been undone already**
* **Undos performed as described earlier**

**Other ARIES Features**

* **Recovery Independence**
  + **Pages can be recovered independently of others**
    - **E.g. if some disk pages fail they can be recovered from a backup while other pages are being used**
* **Savepoints:**
  + **Transactions can record savepoints and roll back to a savepoint**
    - **Useful for complex transactions**
    - **Also used to rollback just enough to release locks on deadlock**
* **Fine-grained locking:**
  + **Index concurrency algorithms that permit tuple level locking on indices can be used**
    - **These require logical undo, rather than physical undo, as in advanced recovery algorithm**
* **Recovery optimizations: For example:**
  + **Dirty page table can be used to prefetch pages during redo**
  + **Out of order redo is possible:**
    - **redo can be postponed on a page being fetched from disk, and  
       performed when page is fetched.**
    - **Meanwhile other log records can continue to be processed**

**Topic: Remote Backup Systems:**

* Remote backup systems provide high availability by allowing transaction processing to continue even if the primary site is destroyed.



* **Detection of failure**: Backup site must detect when primary site has failed
  + to distinguish primary site failure from link failure maintain several communication links between the primary and the remote backup.
  + Heart-beat messages
* **Transfer of control**:
  + To take over control backup site first perform recovery using its copy of the database and all the long records it has received from the primary.
    - Thus, completed transactions are redone and incomplete transactions are rolled back.
  + When the backup site takes over processing it becomes the new primary
  + To transfer control back to old primary when it recovers, old primary must receive redo logs from the old backup and apply all updates locally.
* **Time to recover**: To reduce delay in takeover, backup site periodically proceses the redo log records (in effect, performing recovery from previous database state), performs a checkpoint, and can then delete earlier parts of the log.
* **Hot-Spare** configuration permits very fast takeover:
  + Backup continually processes redo log record as they arrive, applying the updates locally.
  + When failure of the primary is detected the backup rolls back incomplete transactions, and is ready to process new transactions.
* Alternative to remote backup: distributed database with replicated data
  + Remote backup is faster and cheaper, but less tolerant to failure
* Ensure durability of updates by delaying transaction commit until update is logged at backup; avoid this delay by permitting lower degrees of durability.
* **One-safe:**commit as soon as transaction’s commit log record is written at primary
  + Problem: updates may not arrive at backup before it takes over.
* **Two-very-safe:** commit when transaction’s commit log record is written at primary and backup
  + Reduces availability since transactions cannot commit if either site fails.
* **Two-safe:** proceed as in two-very-safe if both primary and backup are active. If only the primary is active, the transaction commits as soon as is commit log record is written at the primary.
  + Better availability than two-very-safe; avoids problem of lost transactions in one-safe.