UNIT-5
Recovery System

Failure Classification

- There are various types of failure that may occur in a system
- **Transaction failure:** There are two types of errors that may cause a transaction to fail:
  - **Logical error:** The transaction can no longer continue with its normal execution because of some internal condition, such as bad input, data not found, overflow, or resource limit exceeded.
  - **System error:** The system has entered an undesirable state (e.g., deadlock), as a result of which a transaction cannot continue with its normal execution.
- **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 disk block loses its content as a result of either a head crash or failure during a data-transfer operation
- To determine how the system should recover from failures, we need to identify the failure modes of those devices used for storing data.
- we must consider how these failure modes affect the contents of the database.
- We can then propose algorithms to ensure database consistency and transaction atomicity despite failures
- Recovery algorithms, have two parts:
  - Actions taken during normal transaction processing
    - to ensure that enough information exists to allow recovery from failures.
  - Actions taken after a failure to recover the database contents
    - to a state that ensures database consistency, transaction atomicity, and durability

Recovery and Atomicity

Log Records

- The most commonly used technique for recovery is based on log records
- The most widely used structure for recording database modifications is the log
- The log is a sequence of log records, recording all the update activities in the database.
- An **update log record** describes a single database write, It has the following 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.
  - it is the location on disk of the data item, consisting of the block identifier of the block on which the data item resides and an offset within the block.
  - **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

- We represent an update log record as<Ti,Xj,V1,V2> , indicating that
  - transaction Ti has performed a write on data item Xj .
  - Xj had value V1 before the write and has value V2 after the write.
- Other special log records
  - <Ti start>   Transaction Ti has started.
  - <Ti commit>   Transaction Ti has committed
  - <Ti abort> Transaction Ti has aborted

- to understand the role of these log records in recovery, we need to consider the steps a transaction takes in modifying 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 holding the data item.
  - The database system executes the output operation that writes the data block to disk.

- Let T0 be a transaction that transfers $50 from account A to account B:

```
T0: read(A);
A := A − 50;
write(A);
read(B);
B := B + 50;
write(B).
```

- Let T1 be a transaction that withdraws $100 from account C:

```
T1: read(C);
C := C − 100;
write(C).
```

- State of the log and database corresponding to T0 and T1.

Log	Database
<T_0 start>	
<T_0, A, 1000, 950>	
<T_0, B, 2000, 2050>	
	A = 950
	B = 2050
<T_0 commit>	
<T_1 start>	
<T_1, C, 700, 600>	
	C = 600
<T_1 commit>	

Database Modification

- a transaction creates a log record prior to modifying the database.
- The log records allow the system to undo changes made by a transaction in the event that the transaction must be aborted;
- they allow the system also to redo changes made by a transaction if the transaction has committed
  - but the system crashed before those changes could be stored in the database on disk.
- a transaction modifies the database if it performs an update on a disk buffer, or on the disk itself;
- updates to the private part of main memory do not count as database modifications.

- **deferred-modification technique:**If a transaction does not modify the database until it has committed
    - need to make local copies of all updated data items
- **Immediate modification technique:** If database modifications occur while the transaction is still active

Concurrency Control and Recovery

- If the concurrency control scheme allows a data item X that has been modified by a transaction T1 to be further modified by another transaction T2 before T1 commits,
    - then undoing the effects of T1 by restoring the old value of X (before T1 updated X) would also undo the effects of T2.
- To avoid such situations, recovery algorithms usually require that if a data item has been modified by a transaction, no other transaction can modify the data item until the first transaction commits or aborts
- This requirement can be ensured by acquiring an exclusive lock on any updated data item and holding the lock until the transaction commits

Transaction Commit

- A transaction is said to have committed when its commit log record is output to stable storage
    - All previous log records of the transaction must have been output already
- there is enough information in the log to ensure that even if there is a system crash, the updates of the transaction can be redone
- If a system crash occurs before a log record<Ti commit> is output to stable storage, transaction Ti will be rolled back

- **Undo and Redo of Transactions**
    - **undo**(T_i) -- restores the value of all data items updated by T_i to their old values, going backwards from the last log record for T_i
        - Each time a data item X is restored to its old value V a special log record <T_i , X, V> is written out
        - When undo of a transaction is complete, a log record <T_i **abort**> is written out.
    - **redo**(T_i) -- sets the value of all data items updated by T_i to the new values, going forward from the first log record for T_i
        - No logging is done in this case

- When recovering after failure:
  - Transaction T_i needs to be undone if the log
    - Contains the record $<T_i$ **start**$>$,
    - But does not contain either the record $<T_i$ **commit**$>$ *or* $<T_i$ **abort**$>$.
  - Transaction T_i needs to be redone if the log
    - Contains the records $<T_i$ **start**$>$
    - And contains the record $<T_i$ **commit**$>$ *or* $<T_i$ **abort**$>$
- Suppose that transaction T_i was undone earlier and the $<T_i$ **abort**$>$ record was written to the log, and then a failure occurs,
- On recovery from failure transaction T_i is redone
  - Such a **redo** redoes all the original actions of transaction T_i *including the steps that restored old values*
    - Known as **repeating history**
    - Seems wasteful, but simplifies recovery greatly

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

$<T_0$ start>	$<T_0$ start>	$<T_0$ start>
$<T_0, A, 1000, 950>$	$<T_0, A, 1000, 950>$	$<T_0, A, 1000, 950>$
$<T_0, B, 2000, 2050>$	$<T_0, B, 2000, 2050>$	$<T_0, B, 2000, 2050>$
	$<T_0$ commit>	$<T_0$ commit>
	$<T_1$ start>	$<T_1$ start>
	$<T_1, C, 700, 600>$	$<T_1, C, 700, 600>$
		$<T_1$ commit>
(a)	(b)	(c)

Recovery actions in each case above are:

(a) undo (T_0): B is restored to 2000 and A to 1000, and log records $<T_0$, B, 2000>, $<T_0$, A, 1000>, $<T_0$, **abort**> are written out

(b) redo (T_0) and undo (T_1): *A* and *B* are set to 950 and 2050 and C is restored to 700. Log records $<T_1$, C, 700>, $<T_1$, **abort**> are written out.

(c) redo (T_0) and redo (T_1): A and B are set to 950 and 2050 respectively. Then *C* is set to 600

Checkpoints

- Redoing/undoing all transactions recorded in the log can be very slow
  - Processing the entire log is time-consuming if the system has run for a long time
  - We might unnecessarily redo transactions which have already output their updates to the database.
- Streamline recovery procedure by periodically performing **checkpointing**
  1. Output all log records currently residing in main memory onto stable storage.
  2. Output all modified buffer blocks to the disk.
  3. Write a log record < **checkpoint** *L*> onto stable storage where *L* is a list of all transactions active at the time of checkpoint.
  4. All updates are stopped while doing checkpointing

- During recovery we need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i.
  - Scan backwards from end of log to find the most recent <**checkpoint** *L*> record
  - Only transactions that are in *L* or started after the checkpoint need to be redone or undone
  - Transactions that committed or aborted before the checkpoint already have all their updates output to stable storage.
- Some earlier part of the log may be needed for undo operations
  - Continue scanning backwards till a record $<T_i$ **start**$>$ is found for every transaction T_i in *L*.
  - Parts of log prior to earliest $<T_i$ **start**$>$ record above are not needed for recovery, and can be erased whenever desired.

Recovery Algorithm

- recovery algorithm using log records for recovery from transaction failure and
- a combination of the most recent checkpoint and
- log records to recover from a system crash.

- **Logging** (during normal operation):
  - <T_i **start**> at transaction start
  - <T_i, X_j, V_1, V_2> for each update, and
  - <T_i **commit**> at transaction end

Transaction Rollback

- **Transaction rollback (during normal operation)**
  - Let T_i be the transaction to be rolled back
  - Scan log backwards from the end, and for each log record of T_i of the form <T_i, X_j, V_1, V_2>
    - Perform the undo by writing V_1 to X_j,
    - Write a log record <T_i , X_j, V_1>
      - such log records are called **compensation log records**
  - Once the record <T_i **start**> is found stop the backward scan and write the log record <T_i **abort**>

Recovery After a System Crash

Recovery actions, when the database system is restarted after a crash, take place in two phases:

- **redo phase:** replay updates of **all** transactions by scanning the log forward from the last checkpoint whether they committed, aborted, or are incomplete
  - The specific steps taken while scanning the log are as follows:
    1. The list of transactions to be rolled back, undo-list, is initially set to the list L in the<checkpoint L> log record.
    2. Whenever a normal log record of the form<Ti,Xj,V1,V2> , or a redo-only log record of the form<Ti,Xj,V2> is encountered, the operation is redone; that is, the value V2 is written to data item Xj .

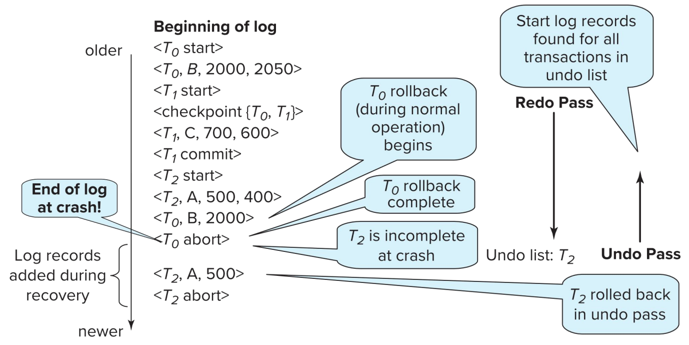
3. Whenever a log record of the form<Ti start> is found, Ti is added to undo-list.

4 . Whenever a log record of the form<Ti abort> or<Ti commit> is found, Ti is removed from undo-list.

- At the end of the redo phase, undo-list contains the list of all transactions that are incomplete,
  - that is, they neither committed nor completed rollback before the crash.

- **Undo phase:** , the system rolls back all transactions in the undo-list.
  - It performs rollback by scanning the log backward from the end
    1. Whenever a log record <T_i, X_j, V_1, V_2> is found where T_i is in undo-list perform same actions as for transaction rollback:
       - perform undo by writing V_1 to X_j.
       - write a log record <T_i , X_j, V_1>
    2. Whenever a log record <T_i **start**> is found where T_i is in undo-list,
       - Write a log record <T_i **abort**>
       - Remove T_i from undo-list
    3. Stop when undo-list is empty
       1. i.e., <T_i **start**> has been found for every transaction in undo-list
  - After undo phase completes, normal transaction processing can commence

Example of Recovery



Log Record Buffering

- **Log record buffering**: log records are buffered in main memory, instead of being output directly to stable storage.
  - Log records are output to stable storage when a block of log records in the buffer is full, or a **log force** operation is executed.
- Log force is performed to commit a transaction by forcing all its log records (including the commit record) to stable storage.
- Several log records can thus be output using a single output operation, reducing the I/O cost.

- The rules below must be followed if log records are buffered:
  - Log records are output to stable storage in the order in which they are created.
  - Transaction T_i enters the commit state only when the log record <T_i **commit**> has been output to stable storage.
  - Before a block of data in main memory is output to the database, all log records pertaining to data in that block must have been output to stable storage.
    - This rule is called the **write-ahead logging** or **WAL** rule
    - WAL only requires undo information to be output

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
- The recovery algorithm supports the **no-force policy**: i.e., updated blocks need not be written to disk when transaction commits
  - **force policy**: requires updated blocks to be written at commit
    - More expensive commit
- The recovery algorithm supports the **steal policy**: i.e., blocks containing updates of uncommitted transactions can be written to disk, even before the transaction commits

---

- 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**.
- **To output a block to disk**
  1. First acquire an exclusive latch on the block
     - Ensures no update can be in progress on the block
  2. Then perform a **log flush**
  3. Then output the block to disk
  4. Finally release the latch on the block

Buffer Management

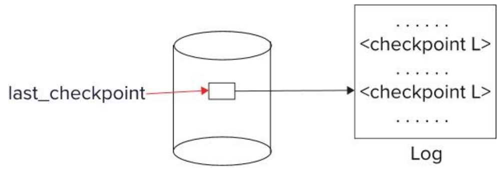
- Database buffer can be implemented either
  - In an area of real main-memory reserved for the database, or
  - In virtual memory
- Implementing buffer in reserved main-memory has drawbacks:
  - Memory is partitioned before-hand between database buffer and applications, limiting flexibility.
  - Needs 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
    - 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
    1. Output the page to database instead of to swap space , if it is modified
    2. 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.

Fuzzy Checkpointing

- To avoid long interruption of normal processing during checkpointing, allow updates to happen during checkpointing
- **Fuzzy checkpointing** is done as follows:
  1. Temporarily stop all updates by transactions
  2. Write a <**checkpoint** *L*> log record and force log to stable storage
  3. Note list *M* of modified buffer blocks
  4. Now permit transactions to proceed with their actions
  5. Output to disk all modified buffer blocks in list *M*
     - blocks should not be updated while being output
     - Follow WAL: all log records pertaining to a block must be output before the block is output
  6. Store a pointer to the **checkpoint** record in a fixed position **last_checkpoint** on disk

- When recovering using a fuzzy checkpoint, start scan from the **checkpoint** record pointed to by **last_checkpoint**
  - Log records before **last_checkpoint** have their updates reflected in database on disk, and need not be redone.
  - Incomplete checkpoints, where system had crashed while performing checkpoint, are handled safely



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**
  - Similar to fuzzy checkpointing

ARIES

- ARIES Incorporates numerous optimizations to reduce overheads during normal processing and to speed up recovery
- Unlike the recovery algorithm described earlier, ARIES
  1. 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
  2. Physiological redo
  3. Dirty page table to avoid unnecessary redos during recovery
  4. 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(Redundant Arrays of Independent Disk), 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

ARIES Data Structures: Log Record

- Each log record contains LSN of previous log record of the same transaction

LSN	TransID	PrevLSN	RedoInfo	UndoInfo

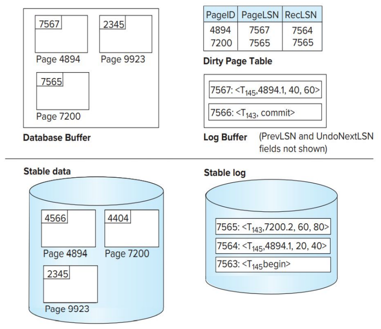
  - LSN in log record may be implicit
- Special redo-only log record called **compensation log record (CLR)** used to log actions taken during recovery that never need to be undone
  - Serves the role of operation-abort log records used in earlier recovery algorithm
  - Has a field UndoNextLSN to note 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
      - Recorded in checkpoints, helps to minimize redo work

- **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

Data Structures used in ARIES



- The log records shown in the figure are prefixed by their LSN
- The data item identifier in a log record is shown in two parts,
- for example,
  - 4894.1 the first identifies the page, and the second part identifies a record within the page
- Each page has an associated PageLSN field.
  - LSN for the last log record that updated page 4894 is 7567.
- By comparing PageLSNs for the pages in the buffer with the PageLSNs for the corresponding pages in stable storage,
  - observe that the DirtyPageTable contains entries for all pages in the buffer that have been modified since they were fetched from stable storage.
- The RecLSN entry in the DirtyPageTable reflects the LSN at the end of the log when the page was added to DirtyPageTable and would be greater than or equal to the PageLSN for that page on stable storage

ARIES Recovery Algorithm

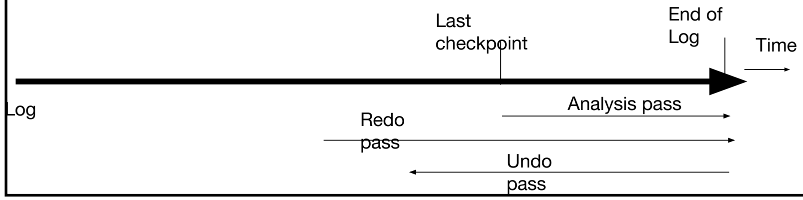
ARIES recovery involves three passes

- 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

Analysis pass (cont.)

- 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:

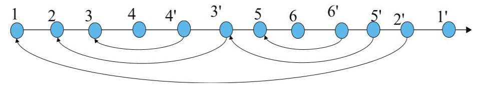
- Scans forward from RedoLSN. Whenever an update log record is found:

  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

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

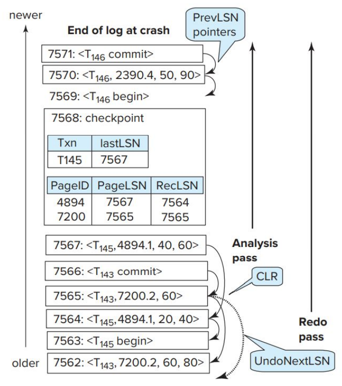


ARIES: Undo Pass

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

Recovery Actions in ARIES



- We assume that the last completed checkpoint pointer on disk points to the checkpoint log record with LSN 7568
- The PrevLSN values in the log records are shown using arrows in the figure,
- the UndoNextLSN value is shown using a dashed arrow for the one compensation log record, with LSN 7565
- The analysis pass would start from LSN 7568, and when it is complete, RedoLSN would be 7564.
- Thus, the redo pass must start at the log record with LSN 7564
- this LSN is less than the LSN of the checkpoint log record, since the ARIES checkpointing algorithm does not flush modified pages to stable storage
- The DirtyPageTable at the end of analysis would include pages 4894, 7200 from the checkpoint log record, and 2390 which is updated by the log record with LSN 7570.

- At the end of the analysis pass, the list of transactions to be undone consists of only T145
- The redo pass for the preceding example starts from LSN 7564 and performs redo of log records whose pages appear in DirtyPageTable
- The undo pass needs to undo only transaction T145,
  - and hence it starts from its LastLSN value 7567 and continues backwards until the record is found at LSN 7563.

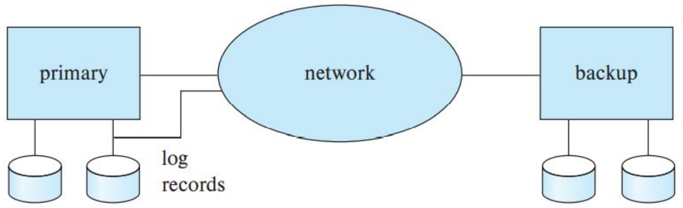
Other ARIES Features

- 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 earlier 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

Remote Backup Systems

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



Architecture of remote backup system.

- **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.
- **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 process the redo log records , 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.