UNIT V

Concurrency Control: Lock-based Protocols, Timestamp-based Protocols, Validation-based Protocols, Multiple Granularity, Multi-version Schemes, Deadlock Handling, Insert and Delete Operations, Weak Levels of Consistency, Concurrency of Index Structures.

Recovery System: Failure Classification, Storage Structure, Recovery and Atomicity, Log-Based Recovery, Recovery with Concurrent Transactions, Buffer Management, Failure with Loss of Nonvolatile Storage, Advanced Recovery Techniques, Remote Backup Systems

CONCURRENCY CONTROL

LOCK-BASED PROTOCOLS:

A locking **protocol** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

A lock is a mechanism to control concurrent access to a data item. Data items can be locked in two modes:

1. *exclusive (X) mode*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
2. *shared (S) mode*. Data item can only be read. S-lock is requested using **lock-S** instruction.

Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

Lock-compatibility matrix:

	S	X
S	true	false
X	false	false

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
- Any number of transactions can hold shared locks on an item,
  - But if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.

Example of a transaction performing locking:

T2: **lock-S***(A)*;
read *(A)*;
unlock*(A)*;
lock-S*(B)*;
read *(B)*;
unlock*(B)*;
display*(A+B)*

- Locking as above is not sufficient to guarantee Serializability — if *A* and *B* get updated in-between the read of *A* and *B*, the displayed sum would be wrong.

Pitfalls of Lock-Based Protocols: Consider the partial schedule

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

Neither *T3* nor *T4* can make progress — executing lock-**S** *(B)* causes *T4* to wait for *T3* to release its lock on *B*, while executing lock-**X** *(A)* causes *T3* to wait for *T4* to release its lock on *A*. Such a situation is called a **deadlock**. To handle a deadlock one of *T3* or *T4* must be rolled back and its locks released. The potential for deadlock exists in most locking protocols.

- **Starvation** is also possible if concurrency control manager is badly designed.
  For example: A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item. The same transaction is repeatedly rolled back due to deadlocks.
- Concurrency control manager can be designed to prevent starvation.

The Two-Phase Locking Protocol:

This is a protocol which ensures conflict-serializable schedules.

Phase 1: Growing Phase: The transaction may obtain locks and transaction may not release locks.

Phase 2: Shrinking Phase: The transaction may release locks and transaction may not obtain locks.

- This protocol assures Serializability. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e. the point where a transaction acquired its final lock).
- Two-phase locking *does not* ensure freedom from deadlocks
- Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking**. Here a transaction must hold all its exclusive locks till it commits/aborts.
- **Rigorous two-phase locking** is even stricter: here *all* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.
- There can be conflict serializable schedules that cannot be obtained if two-phase locking is used.
- However, in the absence of extra information (Ex: ordering of access to data), two-phase locking is needed for conflict Serializability in the following sense:
  Given a transaction *Ti* that does not follow two-phase locking, we can find a transaction *Tj* that uses two-phase locking, and a schedule for *Ti* and *Tj* that is not conflict serializable.

Lock Conversions

- Two-phase locking with lock conversions:
  - ✓ First Phase:
    - can acquire a lock-S on item
    - can acquire a lock-X on item
    - can convert a lock-S to a lock-X (upgrade)

- ✓ Second Phase:
  - can release a lock-S
  - can release a lock-X
  - can convert a lock-X to a lock-S (downgrade)
- This protocol assures Serializability. But still relies on the programmer to insert the various locking instructions.

Automatic Acquisition of Locks: A transaction *T*i issues the standard read/write instruction, without explicit locking calls.

The operation **read** (*D*) is processed as:

```
if Ti has a lock on D then
        read (D)
 else begin
        if necessary wait until no other transaction has a lock-X on D
        grant Ti a lock-S on D;
        read (D)
      end
```

write *(D)* is processed as:

```
if Ti has a lock-X on D then
    write (D)
 else begin
    if necessary wait until no other trans. has any lock on D,
    if Ti has a lock-S on D then
        upgrade lock on D to lock-X
      else
        grant Ti a lock-X on D
      write (D)
 end;
```

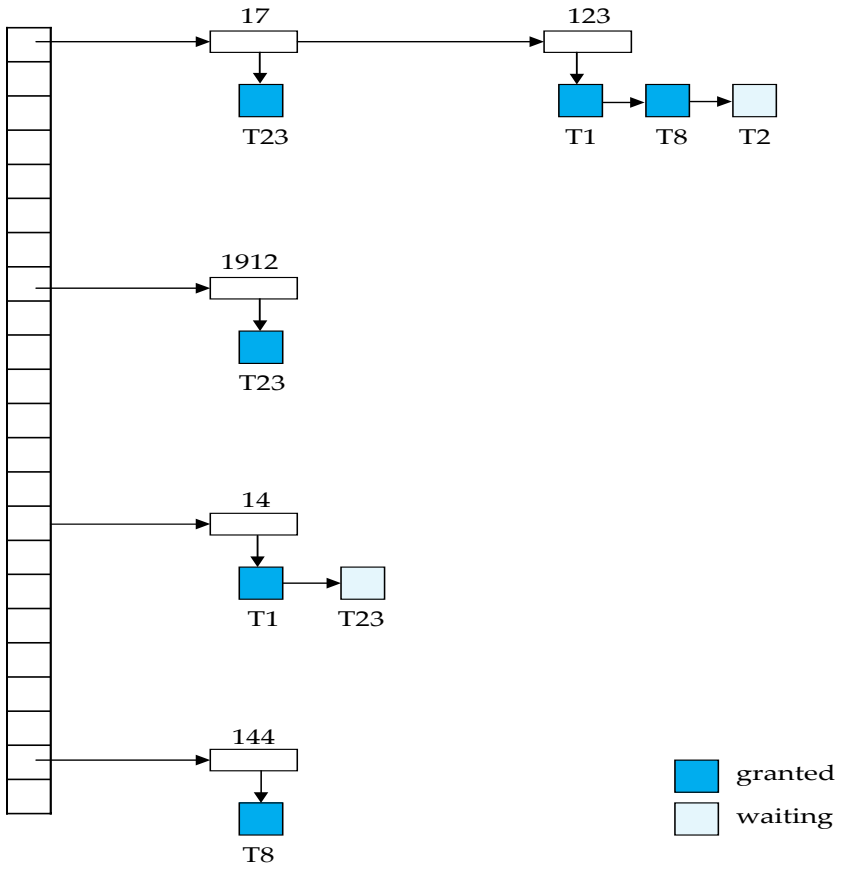
All locks are released after commit or abort

Implementation of Locking:

- A **lock manager** can be implemented as a separate process to which transactions send lock and unlock requests
- The lock manager replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock)
- The requesting transaction waits until its request is answered
- The lock manager maintains a data-structure called a **lock table** to record granted locks and pending requests

Lock Table: The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked. Lock table also records the type of lock granted or requested

- Blue rectangles indicate granted locks, Light Blue rectangles ones indicate waiting requests
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks
- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted
- If transaction aborts, all waiting or granted requests of the transaction are deleted. Lock manager may keep a list of locks held by each transaction, to implement this efficiently



Timestamp-Based Protocols:

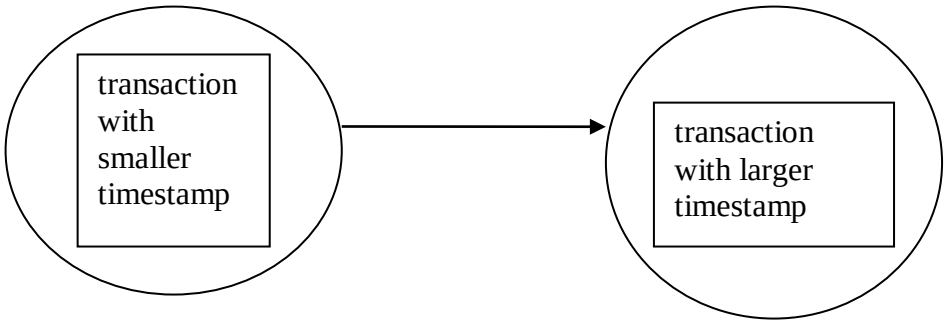
Each transaction is issued a timestamp when it enters the system. If an old transaction *Ti* has time-stamp TS (*Ti*), a new transaction *Tj* is assigned time-stamp TS (*Tj*) such that TS (*Ti*) < TS (*Tj*).

- The protocol manages concurrent execution such that the time-stamps determine the Serializability order.
- In order to assure such behavior, the protocol maintains for each data *Q* two timestamp values:
  - **W-timestamp** (*Q*) is the largest time-stamp of any transaction that executed **write** (*Q*) successfully.
  - **R-timestamp** (*Q*) is the largest time-stamp of any transaction that executed **read** (*Q*) successfully.
- The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.
- Suppose a transaction Ti issues a **read**(*Q*)
  - If TS(*Ti*) ≤ **W**-timestamp(*Q*), then *Ti* needs to read a value of *Q* that was already overwritten.
    - Hence, the **read** operation is rejected, and *Ti* is rolled back.
  - If TS(*Ti*)≥ **W**-timestamp(*Q*), then the **read** operation is executed, and R-timestamp(*Q*) is set to **max**(R-timestamp(*Q*), TS(*Ti*)).
- Suppose that transaction *Ti* issues **write**(*Q*).
  - If TS(*Ti*) < R-timestamp(*Q*), then the value of *Q* that *Ti* is producing was needed previously, and the system assumed that that value would never be produced.
    - Hence, the **write** operation is rejected, and *Ti* is rolled back.
  - If TS(*Ti*) < W-timestamp(*Q*), then *Ti* is attempting to write an obsolete value of *Q*.
    - Hence, this **write** operation is rejected, and *Ti* is rolled back.
  - Otherwise, the **write** operation is executed, and W-timestamp(*Q*) is set to TS(*Ti*).

Example Use of the Protocol: A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5

T1	T2	T3	T4	T5
				read(*X*)
	read(Y)			
read(Y)				
		write(*Y*)		
		write(Z)		
				read(Z)
	read(*X*)			
	abort			
read(*X*)				
		write(Z)		
		abort		
				write(*Y*)
				write(Z)

Correctness of Timestamp-Ordering Protocol: The timestamp-ordering protocol guarantees Serializability since all the arcs in the precedence graph are of the form:



- ✓ Thus, there will be no cycles in the precedence graph

Timestamp protocol ensures freedom from deadlock as no transaction ever waits. But the schedule may not be cascade-free, and may not even be recoverable.

Recoverability and Cascade Freedom

- Problem with timestamp-ordering protocol:
  - Suppose *Ti* aborts, but *Tj* has read a data item written by *Ti*
  - Then *Tj* must abort; if *Tj* had been allowed to commit earlier, the schedule is not recoverable.
  - Further, any transaction that has read a data item written by *Tj* must abort
  - This can lead to cascading rollback --- that is, a chain of rollbacks
- Solution 1:
  - A transaction is structured such that its writes are all performed at the end of its processing
  - All writes of a transaction form an atomic action; no transaction may execute while a transaction is being written
  - A transaction that aborts is restarted with a new timestamp
- Solution 2: Limited form of locking: wait for data to be committed before reading it
- Solution 3: Use commit dependencies to ensure recoverability

Thomas' Write Rule

- Modified version of the timestamp-ordering protocol in which obsolete **write** operations may be ignored under certain circumstances.
- When *Ti* attempts to write data item *Q*, if TS(*Ti*) < W-timestamp(*Q*), then *Ti* is attempting to write an obsolete value of {*Q*}.
    - ***Rather than rolling back Ti as the timestamp ordering protocol would have done, this {write} operation can be ignored***.
- Otherwise this protocol is the same as the timestamp ordering protocol.
- Thomas' Write Rule allows greater potential concurrency.
    - Allows some view-serializable schedules that are not conflict-serializable.

Validation-Based Protocol

Execution of transaction *Ti* is done in three phases.

1) Read and execution phase: Transaction *Ti* writes only to temporary local variables
2) Validation phase: Transaction *Ti* performs a "validation test'' to determine if local variables can be written without violating Serializability.
3) Write phase: If *Ti* is validated, the updates are applied to database; otherwise, Ti is rolled back.

The three phases of concurrently executing transactions can be interleaved, but each transaction must go through the three phases in that order.

Assume for simplicity that the validation and write phase occur together, atomically and serially i.e., only one transaction executes validation/write at a time. Also called as optimistic concurrency control since transaction executes fully in the hope that all will go well during validation

Each transaction Ti has 3 timestamps

1. Start(Ti) : the time when Ti started its execution
2. Validation(Ti): the time when Ti entered its validation phase
3. Finish(Ti) : the time when Ti finished its write phase

Serializability order is determined by timestamp given at validation time, to increase concurrency. Thus TS(Ti) is given the value of Validation(Ti).

This protocol is useful and gives greater degree of concurrency if probability of conflicts is low.

- because the Serializability order is not pre-decided, and
- Relatively few transactions will have to be rolled back.

Validation Test for Transaction *Tj*

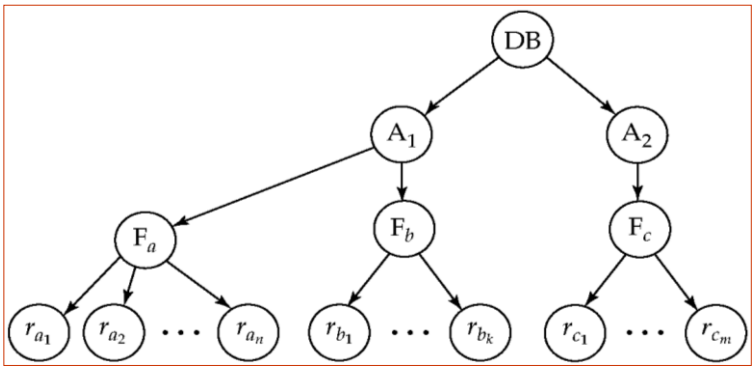
- If for all *Ti* with TS (*Ti*) < TS (*Tj*) either one of the following condition holds:
    - **finish**(*Ti*) < **start**(*Tj*)
    - **start**(*Tj*) < **finish**(*Ti*) < **validation**(*Tj*) **and** the set of data items written by *Ti* does not intersect with the set of data items read by *Tj*.
- Then validation succeeds and *Tj* can be committed. Otherwise, validation fails and *Tj* is aborted.
- *Justification*: Either the first condition is satisfied, and there is no overlapped execution, or the second condition is satisfied and
    - Writes of *Tj* do not affect reads of *Ti* since they occur after *Ti* has finished its reads.
    - Writes of *Ti* do not affect reads of *Tj* since *Tj* does not read any item written by *Ti.*

Schedule Produced by Validation: Example of schedule produced using validation

T14	T15
read(*B*)	
	read(*B*)
	B:= *B-50*
	read(*A*)
	A:= *A+50*
read(*A*)	
(*validate*)	
display (*A*+*B*)	
	(*validate*)
	write (*B*)
	write (*A*)

MULTIPLE GRANULARITY

- Allow data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones
- Can be represented graphically as a tree (but don't confuse with tree-locking protocol)
- When a transaction locks a node in the tree *explicitly*, it *implicitly* locks all nodes descendents in the same mode.
- Granularity of locking (level in tree where locking is done):
  - **fine granularity** (lower in tree): high concurrency, high locking overhead
  - **coarse granularity** (higher in tree): low locking overhead, low concurrency



The levels, starting from the coarsest (top) level are

1. *database*
2. *area*
3. *file*
4. *record*

Intention Lock Modes: In addition to S and X lock modes, there are three additional lock modes with multiple granularity:

1. ***intention-shared*** (IS): Indicates explicit locking at a lower level of the tree but only with shared locks.
2. ***intention-exclusive*** (IX): Indicates explicit locking at a lower level with exclusive or shared locks
3. ***shared and intention-exclusive*** (SIX): The sub tree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.

✓ Intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes.

Compatibility Matrix with Intention Lock Modes: The compatibility matrix for all lock modes is:

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

Multiple Granularity Locking Scheme: Transaction *Ti* can lock a node *Q*, using the following rules:

1. The lock compatibility matrix must be observed.
2. The root of the tree must be locked first, and may be locked in any mode.
3. A node *Q* can be locked by *Ti* in S or IS mode only if the parent of *Q* is currently locked by *Ti* in either IX or IS mode.
4. A node *Q* can be locked by *Ti* in X, SIX, or IX mode only if the parent of *Q* is currently locked by *Ti* in either IX or SIX modes.
5. *Ti* can lock a node only if it has not previously unlocked any node (that is, *Ti* is two-phase).
6. *Ti* can unlock a node *Q* only if none of the children of *Q* are currently locked by *Ti.*

Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order.

MULTIVERSION SCHEMES

- Multiversion schemes keep old versions of data item to increase concurrency.
  - Multiversion Timestamp Ordering
  - Multiversion Two-Phase Locking
- Each successful **write** results in the creation of a new version of the data item written.
- Use timestamps to label versions.
- When a **read**(*Q*) operation is issued, select an appropriate version of *Q* based on the timestamp of the transaction, and return the value of the selected version.
- **read**s never have to wait as an appropriate version is returned immediately.

Multiversion Timestamp Ordering:

- Each data item *Q* has a sequence of versions <*Q1, Q2,...., Qm*>. Each version *Qk* contains three data fields:
  - **Content** -- the value of version *Qk.*
  - **W-timestamp**(*Qk*) -- timestamp of the transaction that created (wrote) version Qk
  - **R-timestamp**(*Qk*) -- largest timestamp of a transaction that successfully read version Qk
- When a transaction *Ti* creates a new version *Qk* of *Q*, Qk's W-timestamp and R-timestamp are initialized to TS(*Ti*).
- R-timestamp of *Qk* is updated whenever a transaction *Tj* reads *Qk*, and TS(*Tj*) > R-timestamp(*Qk*).
- Suppose that transaction *Ti* issues a **read**(*Q*) or **write**(*Q*) operation. Let *Qk* denote the version of *Q* whose write timestamp is the largest write timestamp less than or equal to TS(*Ti*).
  - If transaction *Ti* issues a **read**(*Q*), then the value returned is the content of version Qk.
  - If transaction *Ti* issues a **write**(*Q*)
    - if TS(*Ti*) < R-timestamp(*Qk*), then transaction *Ti* is rolled back.
    - if TS(*Ti*) = W-timestamp(*Qk*), the contents of *Qk* are overwritten
    - else a new version of *Q* is created.

- Observe that
  - Reads always succeed
  - A write by *Ti* is rejected if some other transaction *Tj* that (in the serialization order defined by the timestamp values) should read
    Ti's write, has already read a version created by a transaction older than *Ti*.

✓ *This Protocol guarantees Serializability.*

Multiversion Two-Phase Locking:

- Differentiates between read-only transactions and update transactions
- *Update transactions* acquire read and write locks, and hold all locks up to the end of the transaction. That is, update transactions follow rigorous two-phase locking.
  - Each successful **write** results in the creation of a new version of the data item written.
  - Each version of a data item has a single timestamp whose value is obtained from a counter **ts-counter** that is incremented during commit processing.
- *Read-only transactions* are assigned a timestamp by reading current value of **ts-counter** before they start execution; they follow Multiversion timestamp-ordering protocol to perform reads.
- When an update transaction wants to read a data item:
  - It obtains a shared lock on it, and reads the latest version.
- When it wants to write an item
  - It obtains X lock on; it then creates a new version of the item and sets this version's timestamp to ∞.
- When update transaction *Ti* completes, commit processing occurs:
  - *Ti* sets timestamp on the versions it has created to **ts-counter** + 1
  - *Ti* increments **ts-counter** by 1
- Read-only transactions that start after *Ti* increments **ts-counter** will see values updated by *Ti*.
- Read-only transactions that start before *Ti* increments the **ts-counter** will see the value before the updates by *Ti.*
- *Only serializable schedules are produced.*

Implementation Issues

- Creation of multiple versions increases storage overhead
  - Extra tuples
  - Extra space in each tuple for storing version information
- Versions can, however, be garbage collected
  - Ex: if Q has two versions Q5 and Q9, and the oldest active transaction has timestamp > Q9, then Q5 will never be required again

Deadlock Handling

Consider the following two transactions:

*T*1:	write (*X*)	*T*2:	write(*Y*)
	write(*Y*)		write(*X*)

Schedule with deadlock is given as:

T1	T2
lock-X on *X*	
write (*X*)	
	lock-X on *Y*
	write (*X*)
	wait for **lock-X** on *X*
wait for **lock-X** on *Y*	

- System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set to unlock the data item.

Deadlock Prevention: Deadlock prevention protocols ensure that the system will *never* enter into a deadlock state. Some prevention strategies:

- Require that each transaction locks all its data items before it begins execution (pre declaration).
- Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol).

Deadlock Prevention Strategies: Following schemes use transaction timestamps for the sake of deadlock prevention alone.

1. **Wait-die scheme** (non-preemptive): Older transaction may wait for younger one to release data item. Younger transactions never wait for older ones; they are rolled back instead. A transaction may die several times before acquiring needed data item
2. **Wound-wait scheme** (preemptive) : Older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones. May be fewer rollbacks than *wait-die* scheme.

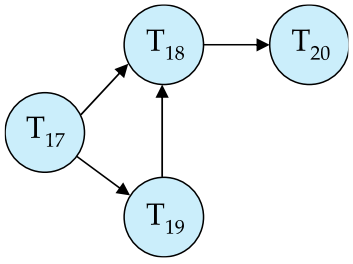
Both in *wait-die* and in *wound-wait* schemes, a rolled back transaction is restarted with its original timestamp. Older transactions thus have precedence over newer ones, and starvation is hence avoided.

3. **Timeout-Based Schemes:** A transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back. Thus deadlocks are not possible. It is simple to implement; but starvation is possible. It is also difficult to determine good value of the timeout interval.

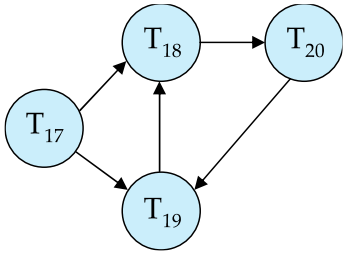
Deadlock Detection: Deadlocks can be described as a *wait-for graph*, which consists of a pair $G = (V,E)$.

- V is a set of vertices (all the transactions in the system)
- E is a set of edges; each element is an ordered pair $Ti \rightarrow Tj$.

- If $Ti \rightarrow Tj$ is in E, then there is a directed edge from Ti to Tj, implying that Ti is waiting for Tj to release a data item.
- When Ti requests a data item currently being held by Tj, then the edge Ti Tj is inserted in the wait-for graph. This edge is removed only when Tj is no longer holding a data item needed by Ti.
- The system is in a deadlock state if and only if the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm periodically to look for cycles.



Wait-for graph without a cycle



Wait-for graph with a cycle

Deadlock Recovery: When a deadlock is detected:

- Some transaction will have to rolled back (made a victim) to break deadlock. Select that transaction as victim that will incur minimum cost.
- Rollback -- determine how far to roll back transaction
  - Total rollback: Abort the transaction and then restart it.
  - More effective to roll back transaction only as far as necessary to break deadlock.
- Starvation happens if same transaction is always chosen as victim. Include the number of rollbacks in the cost factor to avoid starvation

Insert and Delete Operations:

- If two-phase locking is used :
  - A delete operation may be performed only if the transaction deleting the tuple has an exclusive lock on the tuple to be deleted.
  - A transaction that inserts a new tuple into database is given an X-mode lock on the tuple
- Insertions and deletions can lead to the **phantom phenomenon**.
  - A transaction that scans a relation (Ex: find sum of balances of all accounts in Perryridge) and a transaction that inserts a tuple in the relation (Ex: insert a new account at Perryridge) conflict (conceptually) in spite of not accessing any tuple in common.
  - If only tuple locks are used, non-serializable schedules can result
    - Ex: the scan transaction does not see the new account, but reads some other tuple written by the update transaction
- The transaction scanning the relation is reading information that indicates what tuples the relation contains, while a transaction inserting a tuple updates the same information. The information should be locked.
- One solution:
  - Associate a data item with the relation, to represent the information about what tuples the relation contains.
  - Transactions scanning the relation acquire a shared lock in the data item,
  - Transactions inserting or deleting a tuple acquire an exclusive lock on the data item.
- Above protocol provides very low concurrency for insertions/deletions.
- Index locking protocols provide higher concurrency while preventing the phantom phenomenon, by requiring locks on certain index buckets.

Index Locking Protocol: According to this protocol every relation must have at least one index. A transaction can access tuples only after finding them through one or more indices on the relation

- A transaction *Ti* that performs a lookup must lock all index leaf nodes that it accesses, in S-mode
  - Even if the leaf node does not contain any tuple satisfying the index lookup
- A transaction *Ti* that inserts, updates or deletes a tuple *ti* in a relation *r*
  - must update all indices to *r*
  - must obtain exclusive locks on all index leaf nodes affected by the insert/update/delete
- The rules of the two-phase locking protocol must be observed
- Guarantees that phantom phenomenon won't occur

Concurrency in Index Structures: Indices are unlike other database items in that their only job is to help in accessing data.

- ✓ Index-structures are typically accessed very often, much more than other database items.
  - Treating index-structures like other database items, Ex: by 2-phase locking of index nodes can lead to low concurrency.
- ✓ There are several index concurrency protocols where locks on internal nodes are released early, and not in a two-phase fashion.

- It is acceptable to have nonserializable concurrent access to an index as long as the accuracy of the index is maintained.
  - In particular, the exact values read in an internal node of a B^+-tree are irrelevant so long as we land up in the correct leaf node.

✓ Example of index concurrency protocol:

✓ Use **crabbing** instead of two-phase locking on the nodes of the B^+-tree, as follows. During search/insertion/deletion:
- First lock the root node in shared mode.
- After locking all required children of a node in shared mode, release the lock on the node.
- During insertion/deletion, upgrade leaf node locks to exclusive mode.
- When splitting or coalescing requires changes to a parent, lock parent in exclusive mode.

✓ Above protocol can cause excessive deadlocks
- Searches coming down the tree deadlock with updates going up the tree
- Can abort and restart search, without affecting transaction

✓ Better protocols are available; such as the B-link tree protocol
- Intuition: Release lock on parent before acquiring lock on child and deal with changes that may have happened between lock release and acquire

Weak Levels of Consistency

➢ **Degree-two consistency:** differs from two-phase locking in that S-locks may be released at any time, and locks may be acquired at any time
- X-locks must be held till end of transaction
- Serializability is not guaranteed, programmer must ensure that no erroneous database state will occur.

➢ **Cursor stability**:
- For reads, each tuple is locked, read, and lock is immediately released
- X-locks are held till end of transaction
- Special case of degree-two consistency

Weak Levels of Consistency in SQL: SQL allows non-serializable executions

✓ **Serializable:** is the default

✓ **Repeatable read**: allows only committed records to be read, and repeating a read should return the same value (so read locks should be retained)
- However, the phantom phenomenon need not be prevented
  - T1 may see some records inserted by T2, but may not see others inserted by T2

✓ **Read committed**: same as degree two consistency, but most systems implement it as cursor-stability

✓ **Read uncommitted**: allows even uncommitted data to be read

In many database systems, read committed is the default consistency level and has to be explicitly changed to serializable when required as:

n **set isolation level serializable**

RECOVERY SYSTEM

Failure Classification: Failures are basically classified into three. They are:

1. **Transaction Failure**:
   - ✓ **Logical errors**: Transaction cannot complete due to some internal error condition
   - ✓ **System errors**: The database system must terminate an active transaction due to an error condition (Ex: deadlock)
2. **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
3. **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

Recovery Algorithms: Recovery algorithms are techniques to ensure database consistency and transaction atomicity and durability despite failures. Recovery algorithms have two parts:

- ➢ Actions taken during normal transaction processing to ensure enough information exists to recover from failures
- ➢ Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability

Storage Structure:

1. **Volatile storage**: Does not survive system crashes. Examples: main memory, cache memory
2. **Nonvolatile storage**: Survives system crashes. Examples: disk, tape, flash memory
3. **Stable storage**: A mythical form of storage that survives all failures. Approximated by maintaining multiple copies on distinct nonvolatile media.

Stable-Storage Implementation

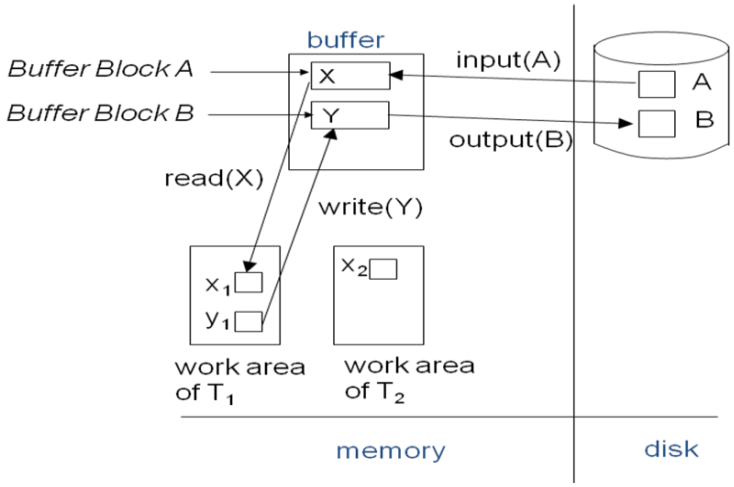
- ➢ Maintain multiple copies of each block on separate disks
  - o Copies can be at remote sites to protect against disasters such as fire or flooding.
- ➢ Failure during data transfer can still result in inconsistent copies: Block transfer can result in
  - o Successful completion
  - o ***Partial Failure***: destination block has incorrect information
  - o ***Total Failure***: destination block was never updated
- ➢ Protecting storage media from failure during data transfer:
  - o Execute output operation as follows (assuming two copies of each block):
    - ▪ Write the information onto the first physical block.
    - ▪ When the first write successfully completes, write the same information onto the second physical block.
    - ▪ The output is completed only after the second write successfully completes.
- ➢ Copies of a block may differ due to failure during output operation. To recover from failure:
  - o First find inconsistent blocks:
    - ▪ **Expensive solution**: Compare the two copies of every disk block.
    - ▪ **Better solution**: Record in-progress disk writes on non-volatile storage. Use this information during recovery to find blocks that may be inconsistent, and only compare copies of these. Used in hardware RAID systems
  - o If either copy of an inconsistent block is detected to have an error (bad checksum), overwrite it by the other copy. If both have no error, but are different, overwrite the second block by the first block.

Data Access: In memory, **Physical blocks** are those blocks residing on the disk. **Buffer blocks** are the blocks residing temporarily in main memory.

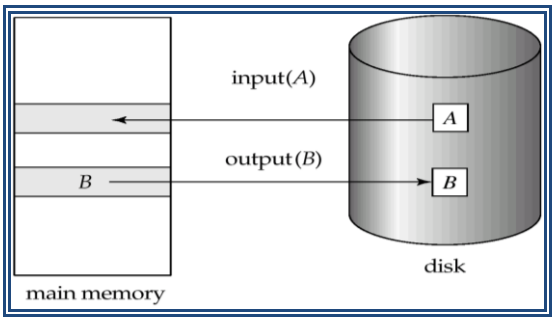
- Block movements between disk and main memory are initiated through the following two operations:
  - **input** (*B*) transfers the physical block *B* to main memory.
  - **output**(*B*) transfers the buffer block *B* to the disk, and replaces the appropriate physical block there.
- Each transaction *Ti* has its private work-area in which local copies of all data items accessed and updated by it are kept.
  - *Ti*'s local copy of a data item *X* is called *xi.*
- We assume, for simplicity, that each data item fits in, and is stored inside, a single block.
- Transaction transfers data items between system buffer blocks and its private work-area using the following operations :
  - **read**(*X*) assigns the value of data item *X* to the local variable *xi*.
  - **write**(*X*) assigns the value of local variable *xi* to data item {*X*} in the buffer block.
  - both these commands may necessitate the issue of an **input**(BX) instruction before the assignment, if the block *BX* in which *X* resides is not already in memory.
- Transactions
  - Perform **read**(*X*) while accessing *X* for the first time;
  - All subsequent accesses are to the local copy.
  - After last access, transaction executes **write**(*X*).
- **output**(*BX*) need not immediately follow **write**(*X*). System can perform the **output** operation when it deems fit.

A buffer block is eventually written out to the disk either because the buffer manager needs the memory space for other purposes or because the database system wishes to reflect the change to *B* on the disk. We shall say that the database system performs a **force-output** of buffer *B* if it issues an output(*B*).

EXAMPLE OF DATA ACCESS



Block Storage Operations:



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 has 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.

Log-Based Recovery:

- Log is a sequence of **log records**, and maintains a record of update activities on the database. A log is kept on stable storage.
- 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* **commi**t> is written.
- We assume for now that log records are written directly to stable storage.
- 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.

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

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

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*0 **start**>		
<*T0,* A, 1000, 950>		
To, B, 2000, 2050		
	A = 950	
	B = 2050	
<*T*0 **commit**>		
<*T*1 **start**>		
<*T*1, C, 700, 600>		
	C = 600	
		BB, *BC*
<*T*1 **commit**>		
		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.** i.e., 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.

<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.

(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. Then *C* is set to 600

Checkpoints:

Problems in recovery procedure include:

1. Searching the entire log is time-consuming
2. We might unnecessarily redo transactions which have already
3. Output their updates to the database.

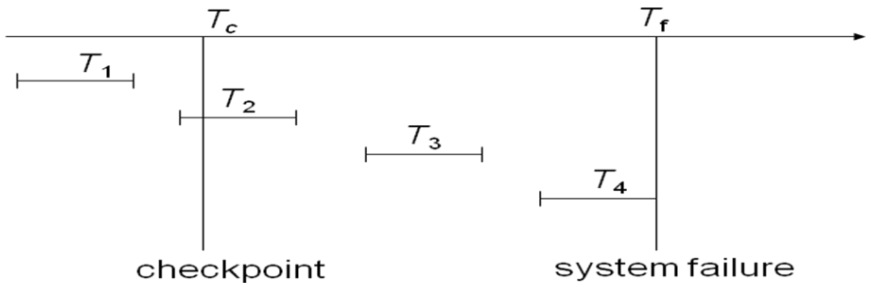
Streamline recovery procedure by periodically performing **checkpointing**

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**> onto stable storage.

During recovery we need to consider only the most recent transaction Ti that started before the checkpoint and transactions that started after *Ti*.

1. Scan backwards from end of log to find the most recent <**checkpoint**> record
2. Continue scanning backwards till a record <*Ti* **start**> is found.
3. Need to only consider the part of log following above **star**t record. Earlier part of log can be ignored during recovery, and can be erased whenever desired.
   - For all transactions (starting from *Ti* or later) with no <*Ti* **commit**>, execute **undo***(Ti).*
   - Scanning forward in the log, for all transactions starting from *Ti* or later with a <*Ti* **commit**>, execute **redo***(Ti).*

Example of Checkpoints



*T*1 can be ignored (updates already output to disk due to checkpoint)

*T*2 and *T*3 redone.

*T*4 undone

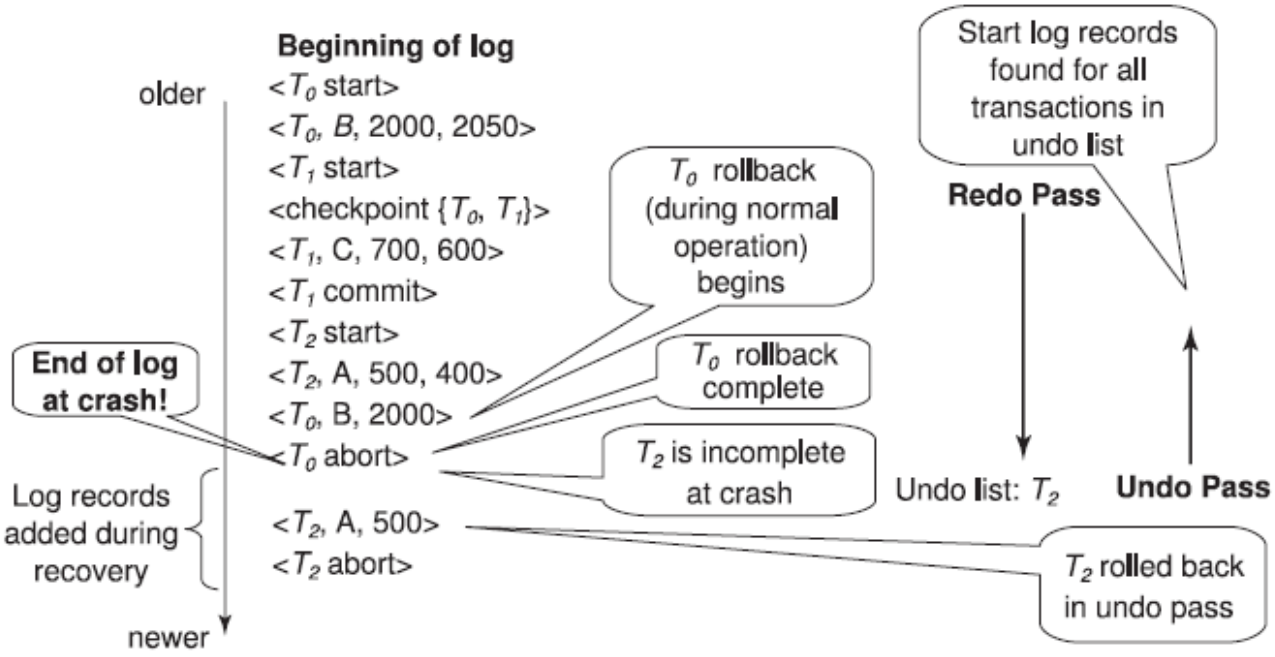
Recovery with Concurrent Transactions: We modify the log-based recovery schemes to allow multiple transactions to execute concurrently. All transactions share a single disk buffer and a single log. A buffer block can have data items updated by one or more transactions

- We assume concurrency control using strict two-phase locking; i.e. the updates of uncommitted transactions should not be visible to other transactions
- Logging is done as described earlier. Log records of different transactions may be interspersed in the log.
- The checkpointing technique and actions taken on recovery have to be changed, since several transactions may be active when a checkpoint is performed.
- The checkpoints are performed as before, except that the checkpoint log record is now of the form < **checkpoint** *L*>, where *L* is the list of transactions active at the time of the checkpoint
    - We assume no updates are in progress while the checkpoint is carried out
- When the system recovers from a crash, it first does the following:
    - Initialize *undo-list* and *redo-list* to empty
    - Scan the log backwards from the end, stopping when the first <**checkpoint** *L*> record is found.

      For each record found during the backward scan:
        - if the record is <*Ti* **commit**>, add *Ti* to *redo-list*
        - if the record is <*Ti* **start**>, then if *Ti* is not in *redo-list*, add *Ti* to *undo-list*
    - For every *Ti* in *L*, if *T*i is not in *redo-list*, add *Ti* to *undo-list*
- At this point *undo-list* consists of incomplete transactions which must be undone, and *redo-list* consists of finished transactions that must be redone.
- Recovery now continues as follows:
    - Scan log backwards from most recent record, stopping when <*Ti* **start**> records have been encountered for every *Ti* in *undo-list*.
        - During the scan, perform **undo** for each log record that belongs to a transaction in *undo-list*.
    - Locate the most recent <**checkpoint** *L*> record.
    - Scan log forwards from the <**checkpoint** *L*> record till the end of the log.
        - During the scan, perform **redo** for each log record that belongs to a transaction on *redo-list*

Example of Recovery: Go over the steps of the recovery algorithm on the following log:

<*T*0 **star**t>
<*T*0, *A*, 0, 10>
<*T*0 **commit**>
<*T*1 **start**> /* Scan at step 1 comes up to here */
<*T*1, *B*, 0, 10>
<*T*2 **start**>
<*T*2, *C*, 0, 10>
<*T*2, *C*, 10, 20>
<checkpoint {*T*1, *T*2}>
<*T*3 **start**>
<*T*3, *A*, 10, 20>
<*T*3, *D*, 0, 10>
<*T*3 **commit**>



Example of logged actions, and actions during recovery.

Buffer Management:

Log Record Buffering: Log records are buffered in main memory, instead of 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 *Ti* enters the commit state only when the log record <*Ti* **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
        - Strictly speaking 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

- ➢ 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
- ➢ 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

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*:
  1. Memory is partitioned before-hand between database buffer and applications, limiting flexibility.
  2. 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!. This is 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.

Failure with Loss of Nonvolatile Storage: A technique similar to checkpointing is 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 DB from most recent dump. Consult 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**

Advanced Recovery Algorithm

Key Features:

- Support for high-concurrency locking techniques, which release locks early.
- Recovery based on "repeating history", whereby recovery executes exactly the same actions as normal processing.
  - including redo of log records of incomplete transactions, followed by subsequent undo
- Supports logical undo.
- Easier to understand/show correctness.

Logical Undo Logging: Operations like B^+ tree insertions and deletions release locks early. They cannot be undone by restoring old values (**physical undo**), since once a lock is released, other transactions may have updated the B^+ tree. Instead, insertions (resp. deletions) are undone by executing a deletion (resp. insertion) operation (known as **logical undo**).

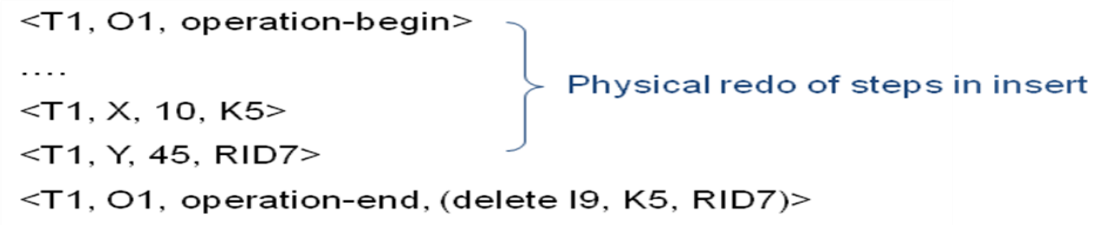
- ✓ For such operations, undo log records should contain the undo operation to be executed. Such logging is called **logical undo logging**, in contrast to **physical undo logging**. Operations are called **logical operations.**
- ✓ Other examples:
  - delete of tuple, to undo insert of tuple, allows early lock release on space allocation information
  - subtract amount deposited, to undo deposit, allows early lock release on bank balance

Physical Redo: Redo information is logged **physically** (i.e., new value for each write) even for operations with logical undo. Physical redo logging does not conflict with early lock release.
Logical redo is very complicated since database state on disk may not be "operation consistent" when recovery starts.

Operation Logging: Operation logging is done as follows:

1. When operation starts, log <T_i, O_j, **operation-begin**>. Here O_j is a unique identifier of the operation instance.
2. While operation is executing, normal log records with physical redo and physical undo information are logged.
3. When operation completes, <T_i, O_j, **operation-end**, U> is logged, where U contains information needed to perform a logical undo information.

Example: insert of (key, record-id) pair (K5, RID7) into index I9



- ✓ If crash/rollback occurs before operation completes: The **operation-end** log record is not found, and the physical undo information is used to undo operation.
- ✓ If crash/rollback occurs after the operation completes:
  - ❖ the **operation-end** log record is found, and in this case
  - ❖ logical undo is performed using U; physical undo information for the operation is ignored.
- ✓ Redo of operation (after crash) still uses physical redo information.

Transaction Rollback: Rollback of transaction T_i is done as follows:
Scan the log backwards,

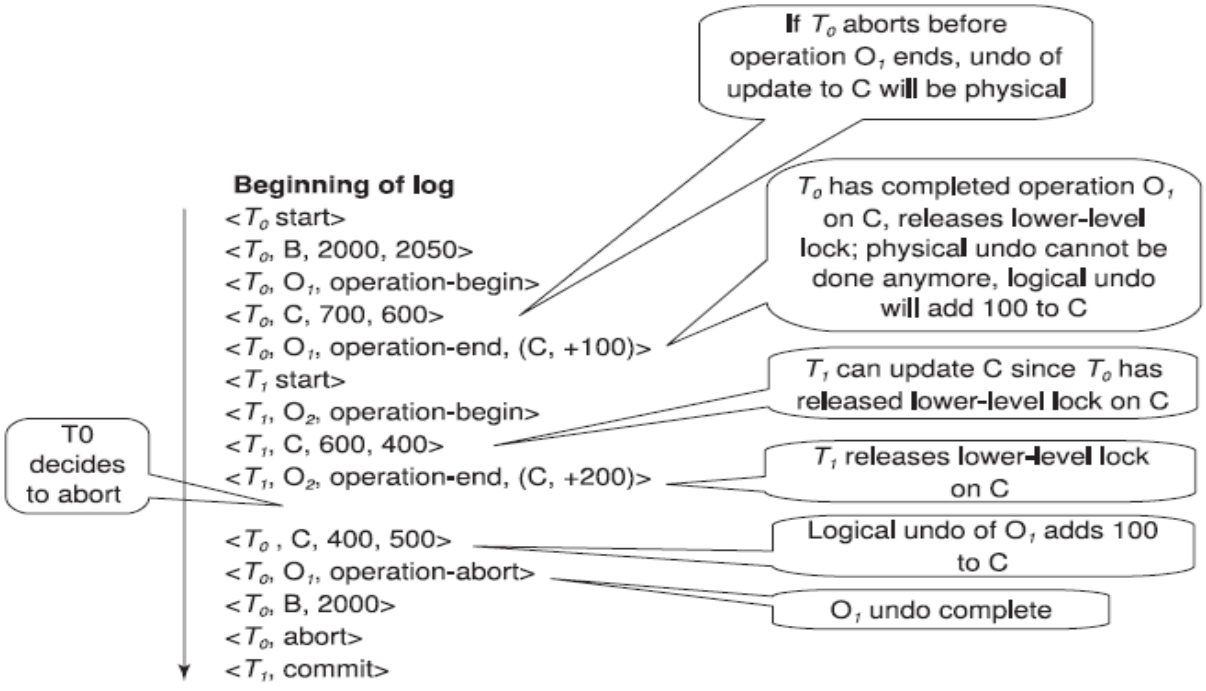
1. If a log record <T_i, X, V_1, V_2> is found, perform the undo and log a special **redo-only log record** <T_i, X, V_1>.
2. If a <T_i, O_j, **operation-end**, U> record is found
   - ✓ Rollback the operation logically using the undo information U.
     - Updates performed during roll back are logged just like during normal operation execution.
     - At the end of the operation rollback, instead of logging an **operation-end** record, generate a record <T_i, O_j, **operation-abort**>.
   - ✓ Skip all preceding log records for T_i until the record <T_i, O_j **operation-begin**> is found
3. If a redo-only record is found ignore it
4. If a <T_i, O_j, **operation-abort**> record is found:
   - ✓ skip all preceding log records for T_i until the record <T_i, O_j, **operation-begi**n> is found.
5. Stop the scan when the record <T_i, start> is found
6. Add a <T_i, **abort**> record to the log

Some points to note:

- Cases 3 and 4 above can occur only if the database crashes while a transaction is being rolled back.
- Skipping of log records as in case 4 is important to prevent multiple rollback of the same operation.

Transaction Rollback Example: Example with a complete and an incomplete operation

```
<T1, start>
<T1, O1, operation-begin>
….
<T1, X, 10, K5>
<T1, Y, 45, RID7>
<T1, O1, operation-end, (delete I9, K5, RID7)>
<T1, O2, operation-begin>
<T1, Z, 45, 70>
                    ← T1 Rollback begins here
<T1, Z, 45>      ← redo-only log record during physical undo (of incomplete O2)
<T1, Y, .., ..>  ← Normal redo records for logical undo of O1
    …
<T1, O1, operation-abort>  ← What if crash occurred immediately after this?
<T1, abort>
```



Transaction rollback with logical undo operations.

Crash Recovery: The following actions are taken when recovering from system crash

1. **Redo Phase**: Scan log forward from last < **checkpoint** *L*> record till end of log
    1. **Repeat history** by physically redoing all updates of all transactions,
    2. Create an undo-list during the scan as follows
        - *undo-list* is set to *L* initially
        - Whenever <T_i **start**> is found T_i is added to *undo-list*
        - Whenever <T_i **commit**> or <T_i **abort**> is found, T_i is deleted from *undo-list*

This brings database to state as of crash, with committed as well as uncommitted transactions having been redone.

Now *undo-list* contains transactions that are **incomplete**, that is, have neither committed nor been fully rolled back.

2. **Undo Phase**: Scan log backwards, performing undo on log records of transactions found in *undo-list*.
    - Log records of transactions being rolled back are processed as described earlier, as they are found
        - Single shared scan for all transactions being undone
    - When <T_i **start**> is found for a transaction T_i in *undo-list*, write a <T_i **abort**> log record.
    - Stop scan when <T_i **start**> records have been found for all T_i in *undo-list*

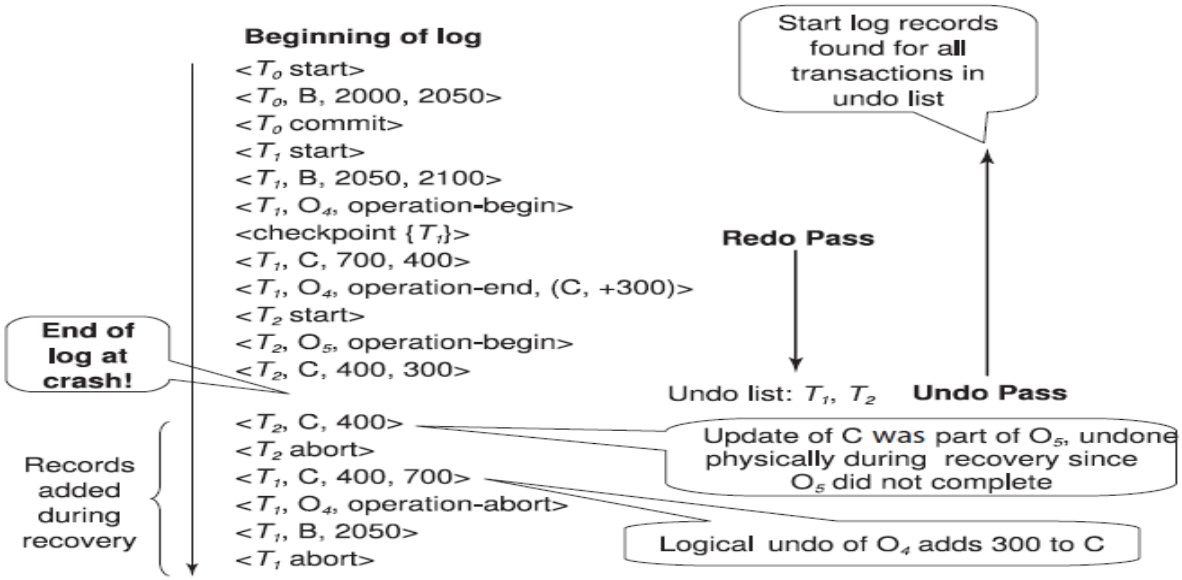
This undoes the effects of incomplete transactions (those with neither **commit** nor **abort** log records). Recovery is now complete.

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.

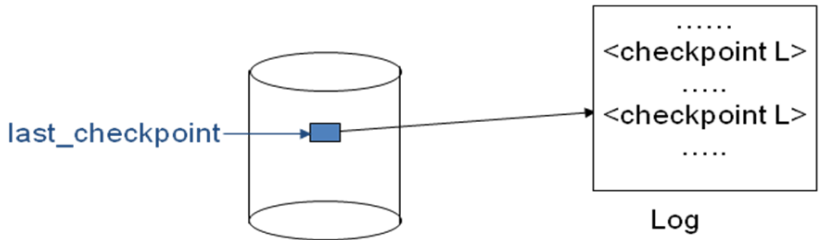
- Fuzzy checkpointing allows transactions to progress while the most time consuming parts of checkpointing are in progress



Failure recovery actions with logical undo operations

Fuzzy 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

ARIES Recovery Algorithm: ARIES is a state of the art recovery method. It incorporates numerous optimizations to reduce overheads during normal processing and to speed up recovery. The "advanced recovery algorithm" we studied earlier is modeled after ARIES, but greatly simplified by removing optimizations

✓ Unlike the advanced 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

ARIES Optimizations:

Physiological redo: Affected page is physically identified, action within page can be logical

❖ Used to reduce logging overheads
- Ex: 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

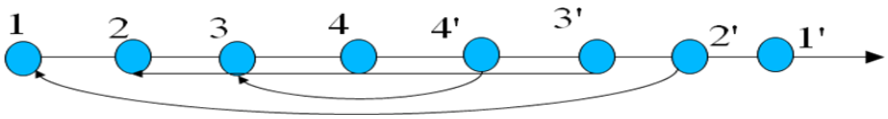
1) **Page LSN:** Each page contains a **PageLSN** which is the LSN of the last log record whose effects are reflected on the page
   - ✓ To update a page:
     - X-latch the page, and write the log record
     - Update the page
     - Record the LSN of the log record in PageLSN
     - Unlock page
   - ✓ 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
2) **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 advanced recovery algorithm
     - Has a field UndoNextLSN to note next (earlier) record to be undone

LSN	TransID	UndoNextLSN	RedoInfo

   - ✓ Records in between would have already been undone
   - ✓ Required to avoid repeated undo of already undone actions



3) **DirtyPage Table:** List of pages in the buffer that have been updated contains,
   - **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

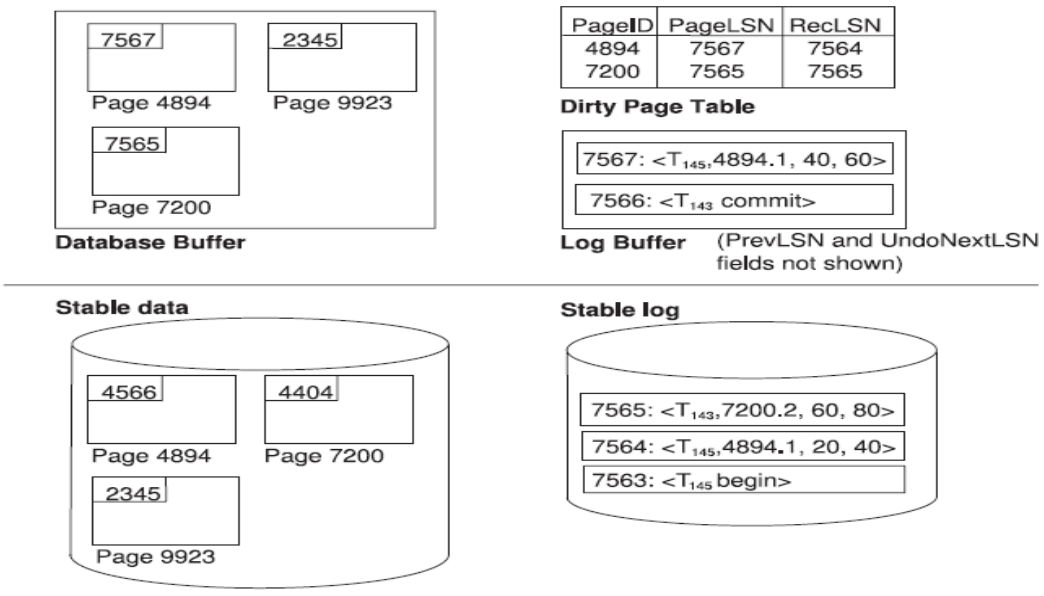


Figure 16.8 Data structures used in ARIES.

4) Checkpoint Log: 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. It can be done frequently

ARIES Recovery Algorithm: ARIES recovery involves three passes

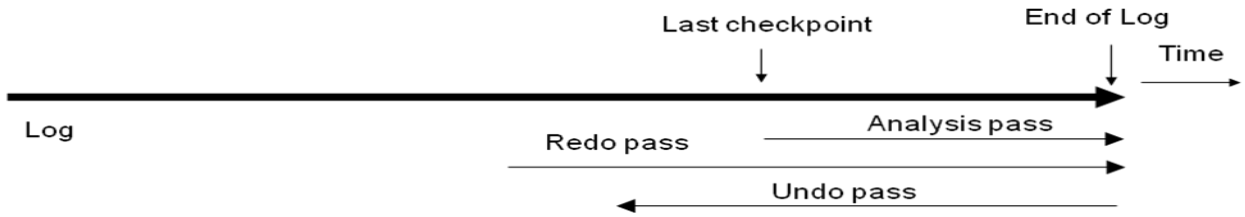
1. ***Analysis Pass:*** Determines
   - Which transactions to undo
   - Which pages were dirty (disk version not up to date) at time of crash
   - RedoLSN: LSN from which redo should start

2. Redo Pass: Repeats history, redoing all actions from RedoLSN

- RecLSN and PageLSNs are used to avoid redoing actions already reflected on page

3. 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



1. **ARIES Analysis Pass:** Analysis determines where redo should start. It 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

- 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

2. **ARIES 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

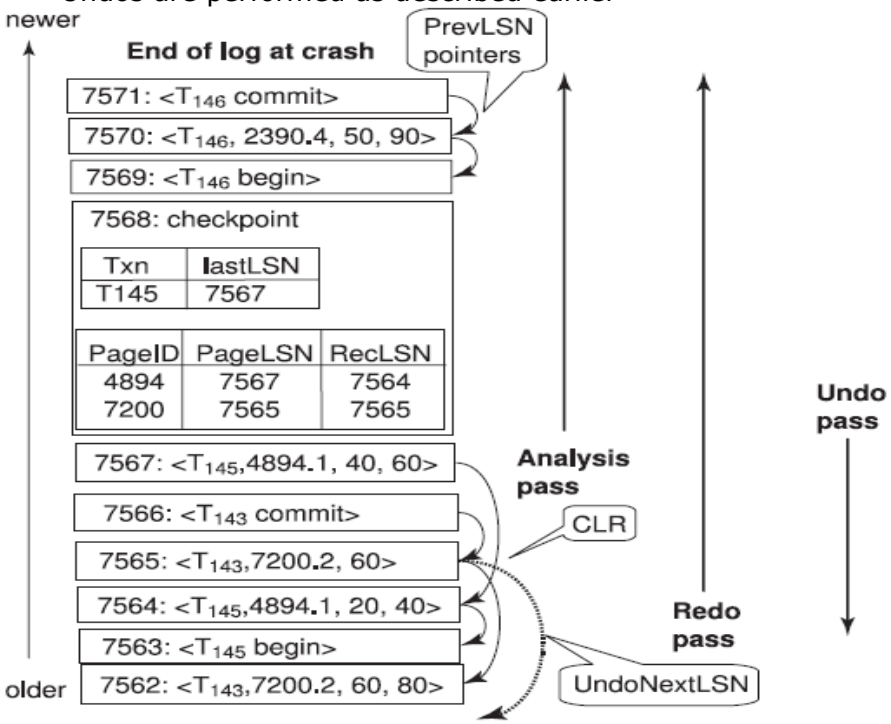
2. Otherwise fetch the page from disk. If the PageLSN of the page fetched from disk is less than the LSN of the log record, redo the log record

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.

3. ARIES Undo Pass: Performs backward scan on log undoing all transaction in undo-list. Undo has to go back till start of earliest incomplete transaction

❖ 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 are performed as described earlier

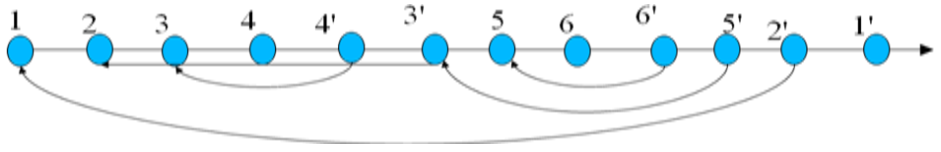


Recovery actions in ARIES.

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 Ex: 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

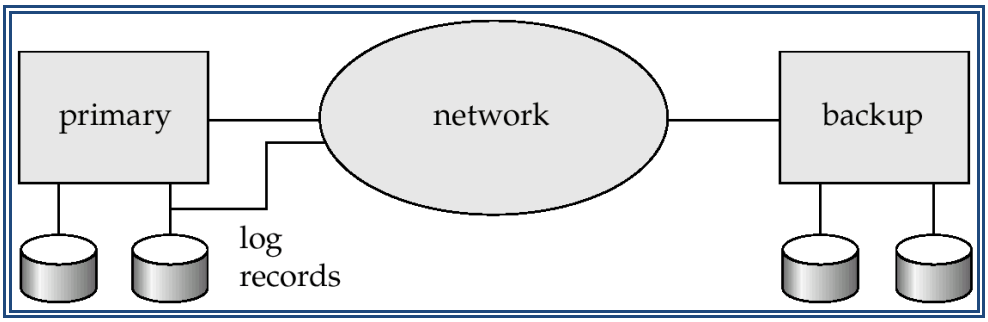


Other ARIES Features:

1. **Recovery Independence**: Pages can be recovered independently of others.
   - Ex: if some disk pages fail they can be recovered from a backup while other pages are being used
2. **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
3. **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
4. **Recovery optimizations:** 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.



- ✓ **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 processes 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 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.

1. **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.
2. **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.
3. **Two-safe:** proceed as in two-very-safe if both primary and backup are active. If only the primary is active, transaction commits as soon as is commit log record is written at primary.
    - Better availability than two-very-safe; avoids problem of lost transactions in one-safe.