LOCK MANAGEMENT

- The part of the DBMS that keeps track of the locks issued to transactions is called the lock manager.
- The lock manager maintains a lock table, with the data object identifier as the key.
- The DBMS also maintains a descriptive entry for each transaction in a transaction table, and among other things, the entry contains a pointer to a list of locks held by the transaction.
- A lock table entry for an object which can be a page or a record contains the information like: the number of transactions currently holding a lock on the object (this can be more than one if the object is locked in shared mode), the nature of the lock (shared or exclusive), and a pointer to a queue of lock requests.

Implementing Lock and Unlock Requests

- According to the Strict 2PL protocol, before a transaction T reads or writes a database object O, it must obtain a shared or exclusive lock on O and must hold on to the lock until it commits or aborts.
- When a transaction needs a lock on an object, it issues a lock request to the lock manager:

1. If a shared lock is requested, the queue of requests is empty, and the object is not currently locked in exclusive mode, the lock manager grants the lock and updates the lock table entry for the object (indicating that the object is locked in shared mode, and incrementing the number of transactions holding a lock by one).
2. If an exclusive lock is requested, and no transaction currently holds a lock on the object (which also implies the queue of requests is empty), the lock manager grants the lock and updates the lock table entry.

- When a transaction aborts or commits, it releases all its locks. When a lock on an object is released, the lock manager updates the lock table entry for the object and examines the lock request at the head of the queue for this object.
- If this request can now be granted, the transaction that made the request is woken up and given the lock.
- Indeed, if there are several requests for a shared lock on the object at the front of the queue, all of these requests can now be granted together.
- Note that if T1 has a shared lock on O, and T2 requests an exclusive lock, T2's request is queued.
- Now, if T3 requests a shared lock, its request enters the queue behind that of T2, even though the requested lock is compatible with the lock held by T1.
- This rule ensures that T2 does not starve, that is, wait indefinitely while a stream of other transactions acquire shared locks and thereby prevent T2 from getting the exclusive lock that it is waiting for.

Atomicity of Locking and Unlocking

- To ensure atomicity of lock and unlock operations when several instances of the lock manager code can execute concurrently, access to the lock table has to be guarded by an operating system synchronization mechanism such as a semaphore.
- If suppose that a transaction requests an exclusive lock.
- The lock manager checks and finds that no other transaction holds a lock on the object and therefore decides to grant the request.
- But in the meantime, another transaction might have requested and received a conflicting lock!
- To prevent this, the entire sequence of actions in a lock request call (checking to see if the request can be granted, updating the lock table, etc.) must be implemented as an atomic operation.

Deadlocks

- Consider the following example
- Transaction T1 sets an exclusive lock on object A, T2 sets an exclusive lock on B, T1 requests an exclusive lock on B and is queued, and T2 requests an exclusive lock on A and is queued.
- Now, T1 is waiting for T2 to release its lock and T2 is waiting for T1 to release its lock. Such a cycle of transactions waiting for locks to be released is called a deadlock.
- Clearly, these two transactions will make no further progress. Worse, they hold locks that may be required by other transactions.
- The DBMS must either prevent or detect (and resolve) such deadlock situations.

Deadlock Prevention

- We can prevent deadlocks by giving each transaction a priority and ensuring that lower priority transactions are not allowed to wait for higher priority transactions (or vice versa).
- One way to assign priorities is to give each transaction a timestamp when it starts up.
- The lower the timestamp, the higher the transaction's priority, that is, the oldest transaction has the highest priority.
- If a transaction Ti requests a lock and transaction Tj holds a conflicting lock, the lock manager can use one of the following two policies:
  - Wait-die: If It has higher priority, it is allowed to wait; otherwise it is aborted.
  - Wound-wait: If It has higher priority, abort Tj; otherwise Ti waits. In the wait-die scheme, lower priority transactions can never wait for higher priority transactions

- In the wound-wait scheme, higher priority transactions never wait for lower priority transactions. In either case no deadlock cycle can develop.
- Transactions having lower timestamp value is having higher priority, this ensures that the oldest transaction will get all the locks that it requires.
- The wait-die scheme is non preemptive; only a transaction requesting a lock can be aborted.

Deadlock Detection

- Deadlocks tend to be rare and typically involve very few transactions.
- This observation suggests that rather than taking measures to prevent deadlocks, it may be better to detect and resolve deadlocks as they arise.
- In the detection approach, the DBMS must periodically check for deadlocks.

- When a transaction Ti is suspended because a lock that it requests cannot be granted, it must wait until all transactions Tj that currently hold conflicting locks release them.
- The lock manager maintains a structure called a waits-for graph to detect deadlock cycles.
- The nodes correspond to active transactions, and there is an arc from Ti to Tj if (and only if) Ti is waiting for Tj to release a lock.
- The lock manager adds edges to this graph when it queues lock requests and removes edges when it grants lock requests.
- The waits-for graph is periodically checked for cycles, which indicate deadlock.
- A deadlock is resolved by aborting a transaction that is on a cycle and releasing its locks; this action allows some of the waiting transactions to proceed.

CONCURRENCY CONTROLWITHOUT LOCKING

- Locking is the most widely used approach to concurrency control in a DBMS, but it is not the only one.
- We now consider some alternative approaches.

Optimistic Concurrency Control

- Locking protocols take a pessimistic approach to conflicts between transactions and use either transaction abort or blocking to resolve conflicts.
- In a system with relatively light contention for data objects, the overhead of obtaining locks and following a locking protocol must nonetheless be paid.
- In optimistic concurrency control, the basic premise is that most transactions will not conflict with other transactions, and the idea is to be as permissive as possible in allowing transactions to execute.

- Transactions proceed in three phases:

  1. Read: The transaction executes, reading values from the database and writing to a private workspace.

  2. Validation: If the transaction decides that it wants to commit, the DBMS checks whether the transaction could possibly have conflicted with any other concurrently executing transaction. If there is a possible conflict, the transaction is aborted; its private workspace is cleared and it is restarted.

  3. Write: If validation determines that there are no possible conflicts, the changes to data objects made by the transaction in its private workspace are copied into the database.

- If, indeed, there are few conflicts, and validation can be done efficiently, this approach should lead to better performance than locking does.
- If there are many conflicts, the cost of repeatedly restarting transactions (thereby wasting the work they've done) will hurt performance significantly.

- Each transaction Ti is assigned a timestamp TS(Ti) at the beginning of its validation phase, and the validation criterion checks whether the timestamp-ordering of transactions is an equivalent serial order.
- For every pair of transactions Ti and Tj such that TS(Ti) < TS(Tj), one of the following conditions must hold:

  1. Ti completes (all three phases) before Tj begins; or
  2. Ti completes before Tj starts its Write phase, and Ti does not write any database object that is read by Tj; or
  3. Ti completes its Read phase before Tj completes its Read phase, and Ti does not write any database object that is either read or written by Tj.

- To validate Tj, we must check to see that one of these conditions holds with respect to each committed transaction Ti such that TS(Ti) < TS(Tj).

- Each of these conditions ensures that Tj's modifications are not visible to Ti.
  - The first condition allows Tj to see some of Ti's changes, but clearly, they execute completely in serial order with respect to each other.
  - The second condition allows Tj to read objects while Ti is still modifying objects, but there is no conflict because Tj does not read any object modified by Ti. Although Tj might overwrite some objects written by Ti, all of Ti's writes precede all of Tj's writes.
  - The third condition allows Ti and Tj to write objects at the same time, and thus have even more overlap in time than the second condition, but the sets of objects written by the two transactions cannot overlap. Thus, no RW, WR, or WW conflicts are possible if any of these three conditions is met.
- Checking these validation criteria requires us to maintain lists of objects read and written by each transaction.

The locking overheads of lock-based approaches are replaced with the overheads of recording read-lists and write-lists for transactions, checking for conflicts, and copying changes from the private workspace.

Timestamp-Based Protocols

- The locking protocols that we have described thus far determine the order between every pair of conflicting transactions at execution time by the first lock that both members of the pair request that involves incompatible modes.
- Another method for determining the serializability order is to select an ordering among transactions in advance.
- The most common method for doing so is to use a timestamp-ordering scheme.

Timestamps

- With each transaction Ti in the system, we associate a unique fixed timestamp, denoted by TS (4).
- This timestamp is assigned by the database system before the transaction Ti starts execution.
- If a transaction Ti has been assigned timestamp TS(Ti), and a new transaction Q enters the system, then TS(4) < TS(4).
- There are two simple methods for implementing this scheme:
- Use the value of the system clock as the timestamp; that is, a transaction's timestamp is equal to the value of the clock when the transaction enters the system.
- Use a logical counter that is incremented after a new timestamp has been assigned; that is, a transaction's timestamp is equal to the value of the counter when the transaction enters the system.

- The timestamps of the transactions determine the serializability order.
- Thus, if TS (4) <TS(T), then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction Ti appears before transaction Q.
- To implement this scheme, we associate with each data item Q two timestamp values:

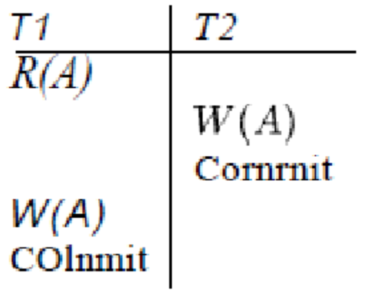
  o W-timestamp (Q) denotes the largest timestamp of any transaction that executed write (Q) successfully.

  o R-timestamp(8) denotes the largest timestamp of any transaction that executed read(Q) successfully.
- These timestamps are updated whenever a new read (Q) or write (Q) instruction is executed.
- **Advantages:** This protocol ensures serializabilty and ensures freedom from deadlock.

Disadvantages: Starvation may occur due to continuously getting aborted and restarting the transaction.

Thomas Write Rule:

Outdated writes are ignored in the Thomas write rule .It works same as Time stamp ordering when R-W,W-R conflicts occurred. It allows more no.of schedules than time stamp ordering as it is flexible than time stamp ordering protocol.



T1	T2
R(A)	
	W(A)
	Cornrnit
W(A)	
COlnmit	

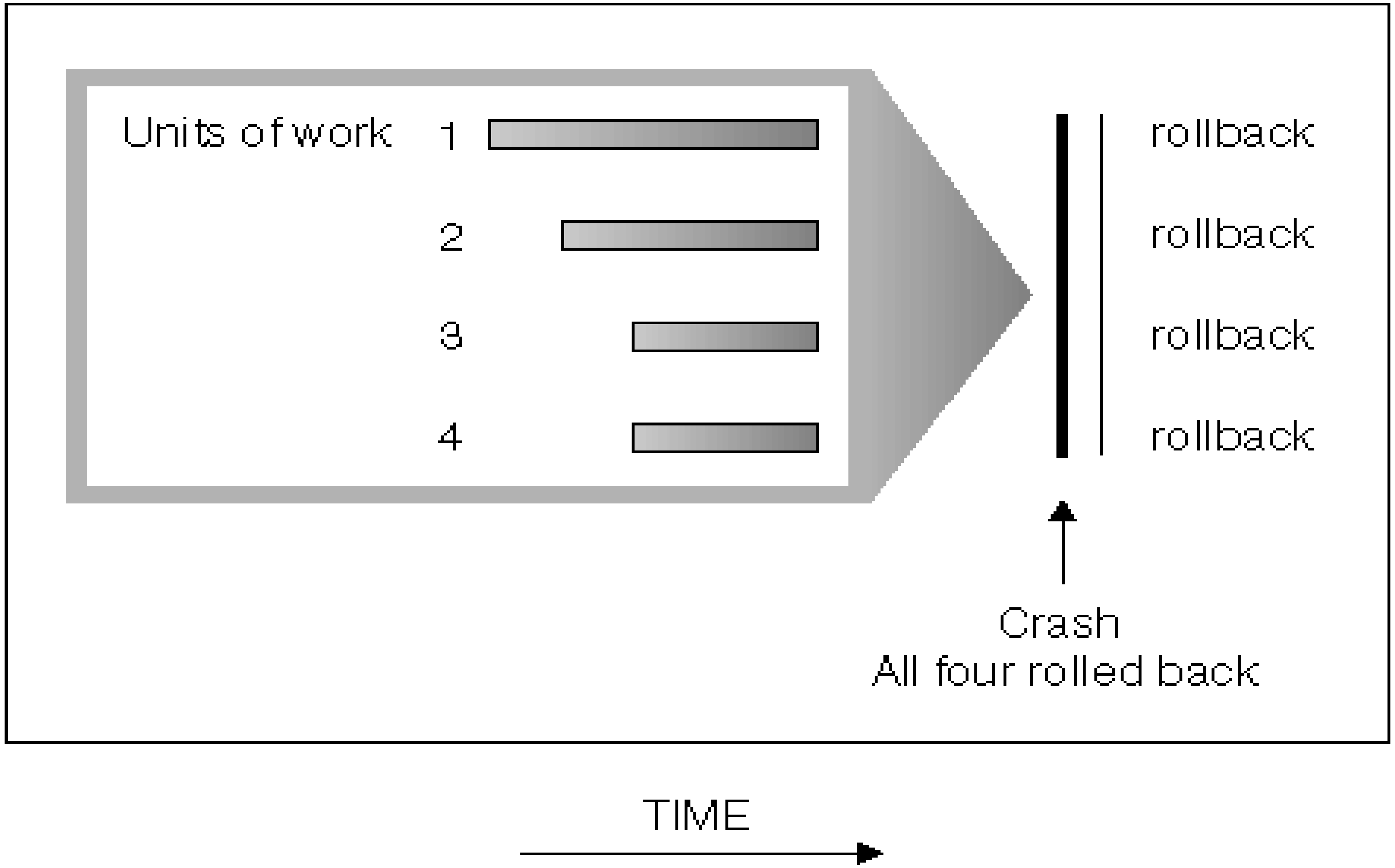
The above schedule is not allowed according to TimeStamp ordering Protocol.But allowed in Thomas Write rule which ignores W(A) inT1.So unnecessary abort operations are avoided.

Crash Recovery

- Transactions (or units of work) against a database can be interrupted unexpectedly.
- If a failure occurs before all of the changes completed, committed, and written to disk, the database is left in an inconsistent and unusable state.
- Crash recovery is the process by which the database is moved back to a consistent and usable state.

This is done by rolling back incomplete transactions and completing committed transactions that were still in memory when the crash occurred

Figure 1. Rolling back units of work (crash recovery)



Algorithm for Recovery and Isolation Exploiting Semantics (ARIES)

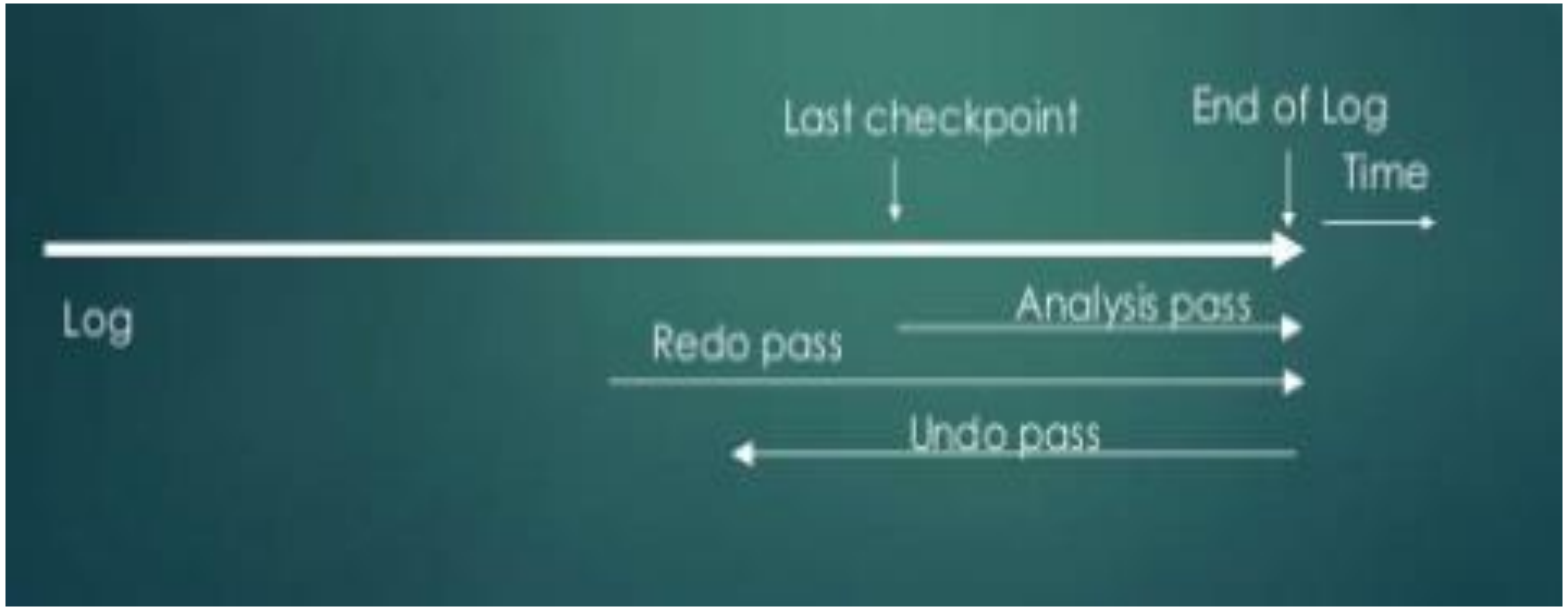
- ARIES stands for "Algorithm for Recovery and Isolation Exploiting Semantics." It was designed to support the needs of *industrial strength* transaction processing systems. ARIES uses logs to record the progress of transactions and their actions which cause changes to recoverable data objects. The log is the source of truth and is used to ensure that committed actions are reflected in the database, and that uncommitted actions are undone.
- Conceptually the log is a single ever-growing sequential file (append-only). Every log record has a unique log sequence number (LSN), and LSNs are assigned in ascending order.

The ARIES recovery procedure consists of three main steps:

- **Analysis**
  The analysis step identifies the dirty (updated) pages in the buffer (Note 6), and the set of transactions active at the time of the crash. The appropriate point in the log where the REDO operation should start is also determined
- **REDO**
  The REDO phase actually reapplies updates from the log to the database. Generally, the REDO operation is applied to only committed transactions. However, in ARIES, this is not the case. Certain information in the ARIES log will provide the start point for REDO, from which REDO operations are applied until the end of the log is reached. In addition, information stored by ARIES and in the data pages will allow ARIES to determine whether the operation to be redone has actually been applied to the database and hence need not be reapplied. Thus only the necessary REDO operations are applied during recovery.

UNDO

During the UNDO phase, the log is scanned backwards and the operations of transactions that were active at the time of the crash are undone in reverse order. The information needed for ARIES to accomplish its recovery procedure includes the log, the Transaction Table, and the Dirty Page



Media Recovery

- Media recovery is based on periodically making a copy of the database.
- When a database object such as a file or a page is corrupted the copy of that is brought up-to-date by using the log.
- Media recovery requires a control file, data files (typically restored from backup), and online and archived redo log files containing changes since the time the data files were backed up. Media recovery is most often used to recover from media failure, such as the loss of a file or disk, or a user error, such as the deletion of the contents of a table.

Recovery from system crash

When the system is restarted after crash the recovery manager proceeds in three phases,

1)Analysis

2)Redo

3)Undo

Few techniques were

Undoing – If a transaction crashes, then the recovery manager may undo transactions i.e. reverse the operations of a transaction.

Deferred update – This technique does not physically update the database on disk until a transaction has reached its commit point. Before reaching commit, all transaction updates are recorded in the local transaction workspace. If a transaction fails before reaching its commit point, it will not have changed the database in any way so UNDO is not needed.

Immediate update – In the immediate update, the database may be updated by some operations of a transaction before the transaction reaches its commit point.

Shadow Paging is a recovery technique that is used to recover databases. In this recovery technique, a database is considered as made up of fixed size of logical units of storage which are referred to as **pages.** pages are mapped into physical blocks of storage, with help of the **page table** which allow one entry for each logical page of database

Backup techniques used were

- **Full database backup –** In this full database including data and database, Meta information needed to restore the whole database, including full-text catalogs are backed up in a predefined time series.
- **Differential backup –** It stores only the data changes that have occurred since last full database backup. When same data has changed many times since last full database backup, a differential backup stores the most recent version of changed data. For this first, we need to restore a full database backup.

- **Transaction log backup –** In this, all events that have occurred in the database, like a record of every single statement executed is backed up. It is the backup of transaction log entries and contains all transaction that had happened to the database. Through this, the database can be recovered to a specific point in time. It is even possible to perform a backup from a transaction log if the data files are destroyed and not even a single committed transaction is lost.