5.7 TWO-PHASE LOCKING TECHNIQUES FOR CONCURRENCY CONTROL

A lock is a variable associated with a data item that describes the status of the item with respect to possible operations that can be applied to it. Generally, there is one lock for each data item in the database. Locks are used as a means of synchronizing the access by concurrent transactions to the database items.

1. Types of Locks and System Lock Tables:

Types of locks used in concurrency control are binary locks, shared/exclusive locks—also known as read/write locks and a certify lock that improves performance of locking protocols.

Binary Locks:

- A binary lock can have two states or values: *locked* and *unlocked* (or 1 and 0).
- A distinct lock is associated with each database item X. If the value of the lock on X is 1, item X cannot be accessed by a database operation that requests the item. If the value of the lock on X is 0, the item can be accessed when requested, and the lock value is changed to 1.
- The current value (or state) of the lock associated with item X is referred to as `lock(X)`.
- Two operations, `lock_item` and `unlock_item`, are used with binary locking.
- A transaction requests access to an item X by first issuing a `lock_item(X)` operation. If `LOCK(X) = 1`, the transaction is forced to wait. If `LOCK(X) = 0`, the transaction locks the item) and the transaction is allowed to access item X.
- When the transaction is through using the item, it issues an `unlock_item(X)` operation, which sets `LOCK(X)` back to 0 (unlocks the item) so that X may be accessed by other transactions. Hence, a binary lock enforces mutual exclusion on the data item.
- A description of the `lock_item(X)` and `unlock_item(X)` operations is shown in Figure 10.

```
lock_item(X):
B:  if LOCK(X) = 0              (*item is unlocked*)
        then LOCK(X) ←1         (*lock the item*)
    else
        begin
        wait (until LOCK(X) = 0
            and the lock manager wakes up the transaction);
        go to B
        end;
unlock_item(X):
    LOCK(X) ← 0;                (* unlock the item *)
    if any transactions are waiting
        then wakeup one of the waiting transactions;
```

Figure 10: Lock and unlock operations for binary locks.

- The `lock_item` and `unlock_item` operations must be implemented as indivisible units that is, no interleaving should be allowed once a lock or unlock operation is started until the operation terminates or the transaction waits.
- In Figure 10, the wait command within the `lock_item(X)` operation is usually implemented by putting the transaction in a waiting queue for item X until X is unlocked and the transaction can be granted access to it. Other transactions that also want to access X are placed in the same queue. Hence, the wait command is considered to be outside the `lock_item` operation.
- A binary-valued variable, `LOCK` is associated with each data item X in the database. Each lock can be a record with three fields: `<Data_item_name, LOCK, Locking_transaction>` plus a queue for transactions that are waiting to access the item. The system needs to maintain only these records for the items that are currently locked in a lock table, which could be organized as a hash file on the item name.
- Items not in the lock table are considered to be unlocked. The DBMS has a lock manager subsystem to keep track of and control access to locks.
- Every transaction must obey the following rules:
  1. A transaction T must issue the operation `lock_item(X)` before any `read_item(X)` or `write_item(X)` operations are performed in T.
  2. A transaction T must issue the operation `unlock_item(X)` after all `read_item(X)` and `write_item(X)` operations are completed in T.
  3. A transaction T will not issue a `lock_item(X)` operation if it already holds the lock on item X.
  4. A transaction T will not issue an `unlock_item(X)` operation unless it already holds the lock on item X.

These rules can be enforced by the lock manager module of the DBMS. Between the `lock_item(X)` and `unlock_item(X)` operations in transaction T, T is said to hold the lock on item X. At most one transaction can hold the lock on a particular item. Thus no two transactions can access the same item concurrently.

Shared/Exclusive (or Read/Write) Locks:

- Several transactions should be allowed to access the same item X if they all access X for reading purposes only. This is because read operations on the same item by different transactions are not conflicting.
- If a transaction is to write an item X, it must have exclusive access to X. For this purpose, a different type of lock, called a multiple-mode lock, is used. In this scheme—called shared/exclusive or

read/write locks—there are three locking operations: `read_lock(X)`, `write_lock(X)`, and `unlock(X)`.

- A lock associated with an item X, `LOCK(X)` has three possible states: read-locked, write-locked, or unlocked.
- A read-locked item is also called share-locked because other transactions are allowed to read the item, whereas a write-locked item is called exclusive-locked because a single transaction exclusively holds the lock on the item.
- Keep track of the number of transactions that hold a shared (read) lock on an item in the lock table, as well as a list of transaction ids that hold a shared lock.
- Each record in the lock table will have four fields: `<Data_item_name, LOCK, No_of_reads, Locking_transaction(s)>`.
- The system needs to maintain lock records only for locked items in the lock table. The value (state) of LOCK is either read-locked or write-locked, suitably coded (if we assume no records are kept in the lock table for unlocked items).
- If `LOCK(X)` = write-locked, the value of locking_transaction(s) is a single transaction that holds the exclusive (write) lock on X. If `LOCK(X)` = read-locked, the value of locking transaction(s) is a list of one or more transactions that hold the shared (read) lock on X.
- The three operations `read_lock(X)`, `write_lock(X)` and `unlock(X)` are described in Figure 11. Each of the three locking operations should be considered indivisible; no interleaving should be allowed once one of the operations is started until either the operation terminates by granting the lock or the transaction is placed in a waiting queue for the item.

```
Read_lock(X):
B: if LOCK(X) = "unlocked"
        then begin LOCK(X) ← "read-locked";
                   no_of_reads(X) ← 1
             end
   else if LOCK(X) = "read-locked"
        then no_of_reads(X) ← no_of_reads(X) + 1
   else begin
        wait (until LOCK(X) = "unlocked" and the lock manager
              wakes up the transaction);
        go to B
         end;
```

```
write_lock(X):
B: if LOCK(X) = "unlocked"
        then LOCK(X) ← "write-locked"
   else begin
          wait (until LOCK(X) = "unlocked" and the lock manager
                wakes up the   transaction);
         go to B
          end;

unlock(X):
if LOCK(X) = "write-locked"
        then begin LOCK(X) ← "unlocked";
                wakeup one of the waiting transactions, if any
             end
else if LOCK(X) = "read-locked"
        then begin
             no_of_reads(X) ← no_of_reads(X) -1;
             if no_of_reads(X) = 0
               then begin LOCK(X) = "unlocked";
                    wakeup one of the waiting transactions, if any
                    end
             end;
```

Figure 11: Locking and unlocking operations for two mode (read/write, or shared/exclusive) locks.

- When the shared/exclusive locking scheme is used, the system must enforce the following rules:

  1. A transaction T must issue the operation `read_lock(X)` or `write_lock(X)` before any `read_item(X)` operation is performed in T.

  2. A transaction T must issue the operation `write_lock(X)` before any `write_item(X)` operation is performed in T.

  3. A transaction T must issue the operation `unlock(X)` after all `read_item(X)` and `write_item(X)` operations are completed in T.

  4. A transaction T will not issue a `read_lock(X)` operation if it already holds a read (shared) lock or a write (exclusive) lock on item X. This rule may be relaxed for downgrading of locks.

  5. A transaction T will not issue a `write_lock(X)` operation if it already holds a read (shared) lock or write (exclusive) lock on item X. This rule may also be relaxed for upgrading of locks, as we discuss shortly.

  6. A transaction T will not issue an `unlock(X)` operation unless it already holds a read (shared) lock or a write (exclusive) lock on item X.

Conversion (Upgrading, Downgrading) of Locks:

A transaction that already holds a lock on item X is allowed under certain conditions to convert the lock from one locked state to another. For example, it is possible for a transaction T to issue a `read_lock(X)` and then later to upgrade the lock by issuing a `write_lock(X)` operation. If T is the only transaction holding a read lock on X at the time it issues the `write_lock(X)` operation, the lock can be upgraded; otherwise, the transaction must wait. It is also possible for a transaction T to issue a `write_lock(X)` and then later to downgrade the lock by issuing a `read_lock(X)` operation.

When upgrading and downgrading of locks is used, the lock table must include transaction identifiers in the record structure for each lock (in the locking_transaction(s) field) to store the information on which transactions hold locks on the item.

Using binary locks or read/write locks in transactions, does not guarantee serializability of schedules on its own. Figure 12 shows an example where the preceding locking rules are followed but a nonserializable schedule may result. This is because in Figure 12(a) the items Y in T_1 and X in T_2 were unlocked too early. This allows a schedule such as the one shown in Figure 12(c) to occur, which is not a serializable schedule and hence gives incorrect results. To guarantee serializability, an additional protocol concerning the positioning of locking and unlocking operations in every transaction must be followed.

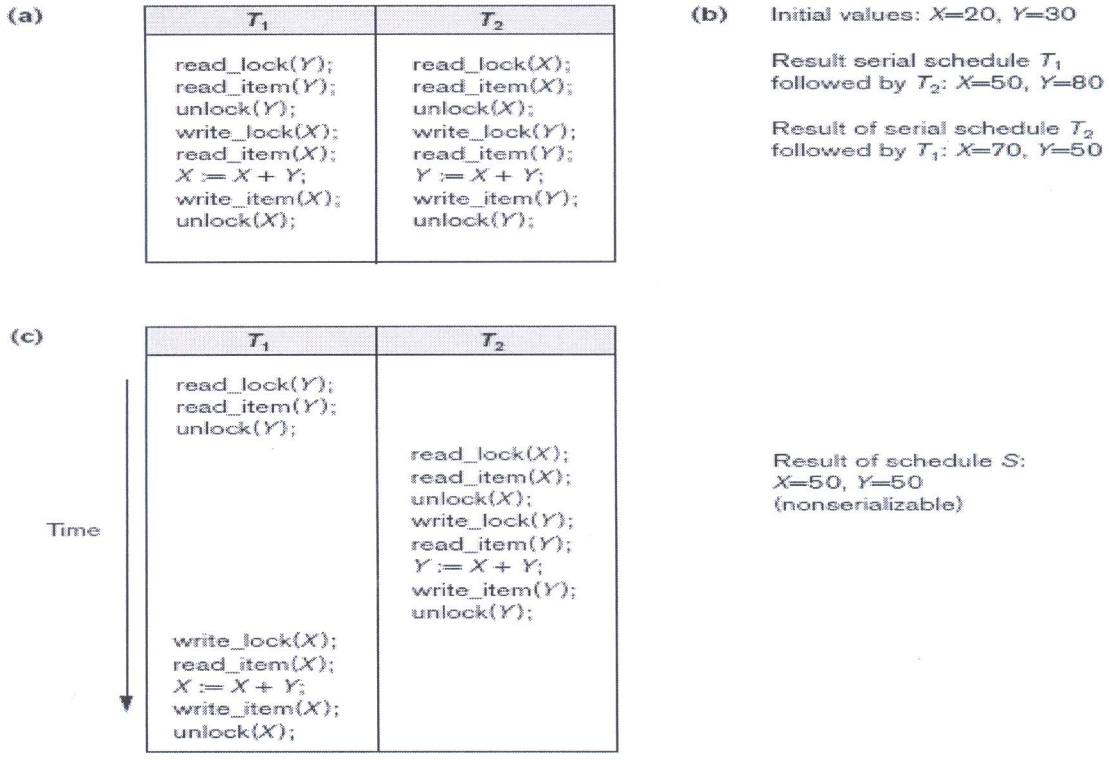

Figure 12: Transactions that do not obey two phase locking.

2. Guaranteeing Serializability by Two-Phase Locking:

- A transaction is said to follow the two-phase locking protocol if all locking operations (read_lock, write_lock) precede the first unlock operation in the transaction.
- Such a transaction can be divided into two phases:
  - an expanding or growing (first) phase, during which new locks on items can be acquired but none can be released;
  - a shrinking (second) phase, during which existing locks can be released but no new locks can be acquired.

  If lock conversion is allowed, then upgrading of locks (from read-locked to write-locked) must be done during the expanding phase, and downgrading of locks (from write-locked to read-locked) must be done in the shrinking phase.
- Transactions T_1 and T_2 in Figure 12(a) do not follow the two-phase locking protocol because the `write_lock(X)` operation follows the `unlock(Y)` operation in T_1, and similarly the `write_lock(Y)` operation follows the `unlock(X)` operation in T_2.
- If two-phase locking is enforced, the transactions can be rewritten as T_1' and T_2', as shown in Figure 13. Now, the schedule shown in Figure 12(c) is not permitted for T_1' and T_2' (with their modified order of locking and unlocking operations) because T_1' will issue its `write_lock(X)` before it unlocks item Y; consequently, when T_2' issues its `read_lock(X)`, it is forced to wait until T_1' releases the lock by issuing `unlock(X)` in the schedule. However, this can lead to deadlock.

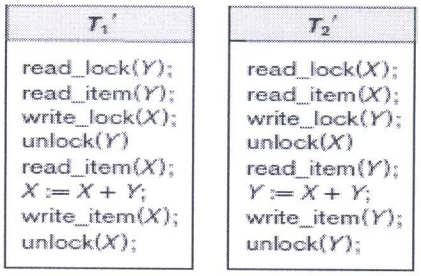

Figure 13: Transactions T_1' and T_2', which are the same as T_1 and T_2 in Figure 21.3 but follow the two-phase locking protocol. Note that they can produce a deadlock.

- If every transaction in a schedule follows the two-phase locking protocol, the schedule is guaranteed to be serializable. The locking protocol, by enforcing two-phase locking rules, also enforces serializability.
- Two-phase locking may limit the amount of concurrency that can occur in a schedule because a transaction T may not be able to release an item X after it is through using it if T must lock an

additional item Y later; or, conversely, T must lock the additional item Y before it needs it so that it can release X. Hence, X must remain locked by T until all items that the transaction needs to read or write have been locked; only then can X be released by T. Meanwhile, another transaction seeking to access X may be forced to wait, even though T is done with X; conversely, if Y is locked earlier than it is needed, another transaction seeking to access Y is forced to wait even though T is not using Y yet. This is the price for guaranteeing serializability of all schedules without having to check the schedules themselves.

- Although the two-phase locking protocol guarantees serializability (that is, every schedule that is permitted is serializable), it does not permit all possible serializable schedules (that is, some serializable schedules will be prohibited by the protocol).

Basic, Conservative, Strict, and Rigorous Two-Phase Locking:

- Conservative 2PL (or static 2PL) requires a transaction to lock all the items it accesses before the transaction begins execution, by predeclaring it's read-set and write-set.
- The read-set of a transaction is the set of all items that the transaction reads, and the write-set is the set of all items that it writes. If any of the predeclared items needed cannot be locked, the transaction does not lock any item; instead, it waits until all the items are available for locking.
- Conservative 2PL is a deadlock-free protocol. But it is difficult to use in practice because of the need to predeclare the read-set and write-set, which is not possible in some situations.
- Strict 2PL guarantees strict schedules where a transaction T does not release any of its `exclusive` (write) locks until after it commits or aborts. Hence, no other transaction can read or write an item that is written by T unless T has committed, leading to a strict schedule for recoverability.
- Strict 2PL is not deadlock-free.
- Rigorous 2PL also guarantees strict schedules. In this variation, a transaction T does not release a*ny of its locks* (exclusive or shared) until after it commits or aborts, and so it is easier to implement than strict 2PL.
- Difference between strict and rigorous 2PL: the former holds write-locks until it commits, whereas the latter holds all locks (read and write).
- Difference between conservative and rigorous 2PL: the former must lock all its items before it starts, so once the transaction starts it is in its shrinking phase; the latter does not unlock any of its items until after it terminates (by committing or aborting), so the transaction is in its expanding phase until it ends.
- Usually the concurrency control subsystem itself is responsible for generating the `read_lock` and `write_lock` requests.

Example: Suppose the system is to enforce the strict 2PL protocol. Then, whenever transaction T issues a `read_item(X)`, the system calls the `read_lock(X)` operation on behalf of T. If the state of `LOCK(X)` is write_locked by some other transaction T′, the system places T in the waiting queue for item X; otherwise, it grants the `read_lock(X)` request and permits the `read_item(X)` operation of T to execute. On the other hand, if transaction T issues a `write_item(X)`, the system calls the `write_lock(X)` operation on behalf of T. If the state of `LOCK(X)` is write_locked or read_locked by some other transaction T′, the system places T in the waiting queue for item X; if the state of `LOCK(X)` is read_locked and T itself is the only transaction holding the read lock on X, the system upgrades the lock to write_locked and permits the `write_item(X)` operation by T. Finally, if the state of `LOCK(X)` is unlocked, the system grants the `write_lock(X)` request and permits the `write_item(X)` operation to execute. After each action, the system must update its lock table appropriately.

- Locking is generally considered to have a high overhead, because every read or write operation is preceded by a system locking request. The use of locks can also cause two additional problems: deadlock and starvation.

3. Dealing with Deadlock and Starvation:

- Deadlock occurs when each transaction T in a set of two or more transactions is waiting for some item that is locked by some other transaction T′ in the set. Hence, each transaction in the set is in a waiting queue, waiting for one of the other transactions in the set to release the lock on an item. But because the other transaction is also waiting, it will never release the lock.
- A simple example is shown in Figure 14(a), where the two transactions T_1' and T_2' are deadlocked in a partial schedule; T_1' is in the waiting queue for X, which is locked by T_2', whereas T_2' is in the waiting queue for Y, which is locked by T1′. Meanwhile, neither T_1' nor T_2' nor any other transaction can access items X and Y.

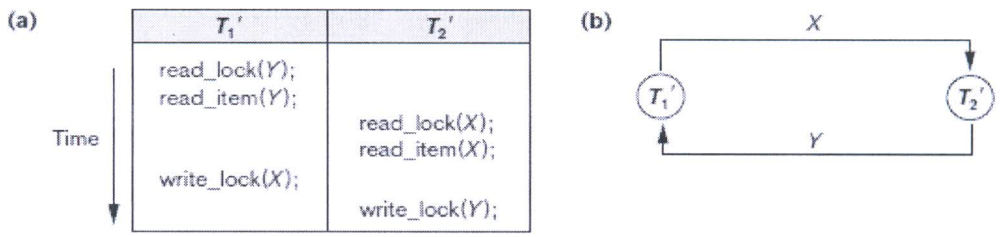

Figure 14: Illustrating the deadlock problem. (a) A partial schedule of T_1' and T_2' that is in a state of deadlock. (b) A wait-for graph for the partial schedule in (a).

Deadlock Prevention Protocols:

- Conservative two-phase locking requires that every transaction lock all the items it needs in advance (which is generally not a practical assumption)—if any of the items cannot be obtained, none of the items are locked. Rather, the transaction waits and then tries again to lock all the items it needs.This solution further limits concurrency.
- A second protocol, which also limits concurrency, involves ordering all the items in the database and making sure that a transaction that needs several items will lock them according to that order. This requires that the programmer (or the system) is aware of the chosen order of the items, which is also not practical in the database context.
- Some of the deadlock prevention techniques use the concept of transaction timestamp TS(T′), which is a unique identifier assigned to each transaction. The timestamps are typically based on the order in which transactions are started; hence, if transaction T_1 starts before transaction T_2, then

  $TS(T_1) < TS(T_2)$.

  The older transaction (which starts first) has the smaller timestamp value.
- Two schemes that prevent deadlock are called `wait-die` and `wound-wait`. Suppose that transaction T_i tries to lock an item X but is not able to because X is locked by some other transaction T_j with a conflicting lock. The rules followed by these schemes are:
  - Wait-die. If $TS(T_i) < TS(T_j)$, then (T_i older than T_j) T_i is allowed to wait; otherwise (Ti younger than T_j) abort T_i (T_i dies) and restart it later with the same timestamp.
  - Wound-wait. If $TS(T_i) < TS(T_j)$, then (T_i older than T_j) abort T_j (T_i wounds T_j) and restart it later with the same timestamp; otherwise (Ti younger than Tj) T_i is allowed to wait.
- In wait-die, an older transaction is allowed to wait for a younger transaction, whereas a younger transaction requesting an item held by an older transaction is aborted and restarted. The wound-wait approach does the opposite: A younger transaction is allowed to wait for an older one, whereas an older transaction requesting an item held by a younger transaction preempts the younger transaction by aborting it. Both schemes end up aborting the younger of the two transactions (the transaction that started later) that may be involved in a deadlock, assuming that this will waste less processing. It can be shown that these two techniques are deadlock-free, since in wait-die, transactions only wait for younger transactions so no cycle is created. Similarly, in wound-wait, transactions only wait for older transactions so no cycle is created. But both techniques may cause some transactions to be aborted and restarted needlessly, even though those transactions may never actually cause a deadlock.
- Protocols that prevent deadlock but do not require timestamps include the **`no waiting`** (NW) and **`cautious waiting`** (CW) algorithms.

- In the no waiting algorithm, if a transaction is unable to obtain a lock, it is immediately aborted and then restarted after a certain time delay without checking whether a deadlock will actually occur or not. In this case, no transaction ever waits, so no deadlock will occur. But this scheme can cause transactions to abort and restart needlessly.
- The cautious waiting algorithm was proposed to try to reduce the number of needless aborts/restarts. Suppose that transaction T_i tries to lock an item X but is not able to do so because X is locked by some other transaction T_j with a conflicting lock. The cautious waiting rule is as follows:

  If Tj is not blocked (not waiting for some other locked item), then T_i is blocked and allowed to wait; otherwise abort T_i.

  Cautious waiting is deadlock-free, because no transaction will ever wait for another blocked transaction. By considering the time b(T) at which each blocked transaction T was blocked, if the two transactions T_i and T_j above both become blocked and T_i is waiting for T_j, then $b(T_i) < b(T_j)$, since T_i can only wait for T_j at a time when T_j is not blocked itself. Hence, the blocking times form a total ordering on all blocked transactions, so no cycle that causes deadlock can occur.

Deadlock Detection:

- In deadlock detection, the system checks if a state of deadlock actually exists. This can happen if different transactions rarely access the same items at the same time, or if transactions are short and each transaction locks only a few items, or if the transaction load is light.
- If transactions are long and each transaction uses many items, or if the transaction load is heavy, it may be advantageous to use a deadlock prevention scheme.
- A simple way to detect a state of deadlock is for the system to construct and maintain **a wait-for graph**. One node is created in the wait-for graph for each transaction that is currently executing. Whenever a transaction T_i is waiting to lock an item X that is currently locked by a transaction T_j, a directed edge ($T_i \rightarrow T_j$) is created in the wait-for graph. When T_j releases the lock(s) on the items that T_i was waiting for, the directed edge is dropped from the wait-for graph. A state of deadlock occurs if and only if the wait-for graph has a cycle.

  This approach has the problem of determining when the system should check for a deadlock. It can be checked for a cycle every time an edge is added to the wait-for graph, but this may cause excessive overhead. Criteria such as the number of currently executing transactions or the period of time several transactions have been waiting to lock items may be used instead to check for a cycle.

Figure 14(b) shows the wait-for graph for the (partial) schedule shown in Figure 14 (a). If the system is in a state of deadlock, some of the transactions causing the deadlock must be aborted. Choosing which transactions to abort is known as victim selection. The algorithm for victim selection should

generally avoid selecting transactions that have been running for a long time and that have performed many updates, and it should try instead to select transactions that have not made many changes (younger transactions).

- Timeouts: A scheme to deal with deadlock is the use of timeouts. In this method, if a transaction waits for a period longer than a system-defined timeout period, the system assumes that the transaction may be deadlocked and aborts it—regardless of whether a deadlock actually exists.

Starvation:

- When locking is used, starvation can occur when a transaction cannot proceed for an indefinite period of time while other transactions in the system continue normally.
- This may occur if the waiting scheme for locked items is unfair in that it gives priority to some transactions over others.
- One solution for starvation is to have a fair waiting scheme, such as using a first-come-first-served queue; transactions are enabled to lock an item in the order in which they originally requested the lock.
- Another scheme allows some transactions to have priority over others but increases the priority of a transaction the longer it waits, until it eventually gets the highest priority and proceeds.
- Starvation can also occur because of victim selection if the algorithm selects the same transaction as victim repeatedly, thus causing it to abort and never finish execution.
- The algorithm can use higher priorities for transactions that have been aborted multiple times to avoid this problem.
- The wait-die and wound-wait schemes avoid starvation, because they restart a transaction that has been aborted with its same original timestamp, so the possibility that the same transaction is aborted repeatedly is slim.

5.8 CONCURRENCY CONTROL BASED ON TIMESTAMP ORDERING

The use of locking, combined with the 2PL protocol, guarantees serializability of schedules. The serializable schedules produced by 2PL have their equivalent serial schedules based on the order in which executing transactions lock the items they acquire. If a transaction needs an item that is already locked, it may be forced to wait until the item is released. Some transactions may be aborted and restarted because of the deadlock problem. A different approach to concurrency control involves using transaction timestamps to order transaction execution for an equivalent serial schedule.

1. **Timestamps:**

- The timestamp of transaction T is referred to as `TS(T)`.

- A timestamp is a unique identifier created by the DBMS to identify a transaction. Timestamp values are assigned in the order in which the transactions are submitted to the system, so a timestamp can be thought of as the transaction start time.
- Concurrency control techniques based on timestamp ordering do not use locks. Hence, deadlocks cannot occur.
- Timestamps can be generated in several ways.
  - Use a counter that is incremented each time its value is assigned to a transaction. The transaction timestamps are numbered 1, 2, 3, ... in this scheme. A computer counter has a finite maximum value, so the system must periodically reset the counter to zero when no transactions are executing for some short period of time.
  - Use the current date/time value of the system clock and ensure that no two timestamp values are generated during the same tick of the clock.

2. The Timestamp Ordering Algorithm for Concurrency Control:

- This scheme enforces the equivalent serial order on the transactions based on their timestamps.
- A schedule in which the transactions participate is then serializable, and the only equivalent serial schedule permitted has the transactions in order of their timestamp values. This is called timestamp ordering (TO).
- Timestamp ordering differs from 2PL, where a schedule is serializable by being equivalent to some serial schedule allowed by the locking protocols. In timestamp ordering, the schedule is equivalent to the particular serial order corresponding to the order of the transaction timestamps.
- The algorithm allows interleaving of transaction operations, but it must ensure that for each pair of conflicting operations in the schedule, the order in which the item is accessed must follow the timestamp order. To do this, the algorithm associates with each database item X two timestamp (TS) values:
  1. `read_TS(X)`. The read timestamp of item X is the largest timestamp among all the timestamps of transactions that have successfully read item X—that is, `read_TS(X) = TS(T)`, where T is the youngest transaction that has read X successfully.
  2. `write_TS(X)`. The write timestamp of item X is the largest of all the timestamps of transactions that have successfully written item X—that is, `write_TS(X) = TS(T)`, where T is the youngest transaction that has written X successfully. Based on the algorithm, T will also be the last transaction to write item X.

Basic Timestamp Ordering (TO): Whenever some transaction T tries to issue a `read_item(X)` or a `write_item(X)` operation, the basic TO algorithm compares the timestamp of T with `read_TS(X)` and `write_TS(X)` to ensure that the timestamp order of transaction execution is not violated. If this order is violated, then transaction T is aborted and resubmitted to the system as a new transaction with a new timestamp. If T is aborted and rolled back, any transaction T_1 that may have used a value written by T must also be rolled back. Similarly, any transaction T_2 that may have used a value written by T_1 must also be rolled back, and so on. This effect is known as cascading rollback and is one of the problems associated with basic TO, since the schedules produced are not guaranteed to be recoverable. An additional protocol must be enforced to ensure that the schedules are recoverable, cascadeless, or strict.

The concurrency control algorithm must check whether conflicting operations violate the timestamp ordering in the following two cases:

1. Whenever a transaction T issues a `write_item(X)` operation, the following check is performed:
   a. If `read_TS(X) > TS(T)` or if `write_TS(X) > TS(T)`, then abort and roll back T and reject the operation. This should be done because some younger transaction with a timestamp greater than `TS(T)`—and hence after T in the timestamp ordering—has already read or written the value of item X before T had a chance to write X, thus violating the timestamp ordering.
   b. If the condition in part (a) does not occur, then execute the `write_item(X)` operation of T and set `write_TS(X)` to `TS(T)`.
2. Whenever a transaction T issues a read_item(X) operation, the following check is performed:
   a. If `write_TS(X) > TS(T)`, then abort and roll back T and reject the operation. This should be done because some younger transaction with timestamp greater than `TS(T)`—and hence after T in the timestamp ordering—has already written the value of item X before T had a chance to read X.
   b. If `write_TS(X)` $\leq$ `TS(T)`, then execute the read_item(X) operation of T and set `read_TS(X)` to the larger of `TS(T)` and the current `read_TS(X)`.

Whenever the basic TO algorithm detects two conflicting operations that occur in the incorrect order, it rejects the later of the two operations by aborting the transaction that issued it. The schedules produced by basic TO are hence guaranteed to be conflict serializable. Deadlock does not occur with timestamp ordering. But cyclic restart (and hence starvation) may occur if a transaction is continually aborted and restarted.

Strict Timestamp Ordering (TO): Strict TO ensures that the schedules are both strict (for easy recoverability) and (conflict) serializable. In this variation, a transaction T issues a `read_item(X)` or

write_item(X) such that TS(T) > write_TS(X) has its read or write operation delayed until the transaction T′ that wrote the value of X (hence TS(T′) = write_TS(X)) has committed or aborted.
To implement this algorithm, it is necessary to simulate the locking of an item X that has been written by transaction T′ until T′ is either committed or aborted. This algorithm does not cause deadlock, since T waits for T′ only if TS(T) > TS(T′).

Thomas's Write Rule: A modification of the basic TO algorithm, known as Thomas's write rule, does not enforce conflict serializability, but it rejects fewer write operations by modifying the checks for the write_item(X) operation as follows:

1. If read_TS(X) > TS(T), then abort and roll back T and reject the operation.
2. If write_TS(X) > TS(T), then do not execute the write operation but continue processing. This is because some transaction with timestamp greater than TS(T)—and hence after T in the timestamp ordering—has already written the value of X. Thus, we must ignore the write_item(X) operation of T because it is already outdated and obsolete. Notice that any conflict arising from this situation would be detected by case (1).
3. If neither the condition in part (1) nor the condition in part (2) occurs, then execute the write_item(X) operation of T and set write_TS(X) to TS(T).

QUESTION BANK:

1. What is the two-phase locking protocol? How does it guarantee serializability?
2. What are some variations of the two-phase locking protocol? Why is strict or rigorous two-phase locking often preferred?
3. Discuss the problems of deadlock and starvation, and the different approaches to dealing with these problems.
4. Describe the wait-die and wound-wait protocols for deadlock prevention.
5. Describe the cautious waiting, no waiting, and timeout protocols for deadlock prevention.
6. What is a timestamp? How does the system generate timestamps?
7. Discuss the timestamp ordering protocol for concurrency control. How does strict timestamp ordering differ from basic timestamp ordering?

5.12 RECOVERY CONCEPTS

1. Recovery Outline and Categorization of Recovery Algorithms:

- Recovery from transaction failures usually means that the database is *restored* to the most recent consistent state before the time of failure. To do this, the system must keep information about the

changes that were applied to data items by the various transactions. This information is typically kept in the **system log**.

- A typical strategy for recovery may be summarized informally as follows:
  1. If there is extensive damage to a wide portion of the database due to catastrophic failure, such as a disk crash, the recovery method restores a past copy of the database that was *backed up* to archival storage (typically tape or other large capacity offline storage media) and reconstructs a more current state by reapplying or *redoing* the operations of committed transactions from the *backed-up* log, up to the time of failure.
  2. When the database on disk is not physically damaged, and a non-catastrophic failure has occurred, the recovery strategy is to identify any changes that may cause an inconsistency in the database. For example, a transaction that has updated some database items on disk but has not been committed needs to have its changes reversed by *undoing* its write operations. It may also be necessary to *redo* some operations in order to restore a consistent state of the database; for example, if a transaction has committed but some of its write operations have not yet been written to disk. For non-catastrophic failure, the recovery protocol does not need a complete archival copy of the database. Rather, the entries kept in the online system log on disk are analyzed to determine the appropriate actions for recovery.

- Two main policies for recovery from non-catastrophic transaction failures: deferred update and immediate update.
- The **deferred update** techniques do not physically update the database on disk until *after* a transaction commits; then the updates are recorded in the database. Before reaching commit, all transaction updates are recorded in the local transaction workspace or in the main memory buffers that the DBMS maintains. Before commit, the updates are recorded persistently in the log file on disk, and then after commit, the updates are written to the database from the main memory buffers. If a transaction fails before reaching its commit point, it will not have changed the database on disk in any way, so UNDO is not needed. It may be necessary to REDO the effect of the operations of a committed database on disk. Hence, deferred update is also known as the NO-UNDO/REDO algorithm.
- In the immediate update techniques, the database may be updated by some operations of a transaction before the transaction reaches its commit point. These operations must also be recorded in the log on disk by force-writing before they are applied to the database on disk, making recovery still possible. If a transaction fails after recording some changes in the database on disk but before reaching its commit point, the effect of its operations on the database must be undone; that is, the transaction must be rolled back. In the general case of immediate update, both undo and redo may be required during

recovery. This technique, known as the UNDO/REDO algorithm, requires both operations during recovery and is used most often in practice. A variation of the algorithm where all updates are required to be recorded in the database on disk before a transaction commits requires undo only, so it is known as the UNDO/NO-REDO algorithm.

- The UNDO and REDO operations are required to be ***idempotent***—that is, executing an operation multiple times is equivalent to executing it just once. The whole recovery process should be idempotent because if the system were to fail during the recovery process, the next recovery attempt might UNDO and REDO certain `write_item` operations that had already been executed during the first recovery process. The result of recovery from a system crash during recovery should be the same as the result of recovering when there is no crash during recovery.

2. Caching (Buffering) of Disk Blocks:

- The recovery process is often closely intertwined with operating system functions— in particular, the buffering of database disk pages in the DBMS main memory cache.
- Multiple disk pages that include the data items to be updated are cached into main memory buffers and then updated in memory before being written back to disk. The caching of disk pages is traditionally an operating system function, but because of its importance to the efficiency of recovery procedures, it is handled by the DBMS by calling low-level operating systems routines.
- In general, it is convenient to consider recovery in terms of the database disk pages (blocks). Typically a collection of in-memory buffers, called the DBMS cache, is kept under the control of the DBMS for the purpose of holding these buffers. A directory for the cache is used to keep track of which database items are in the buffers. This can be a table of `<Disk_page_address, Buffer_location, ... >` entries.
- When the DBMS requests action on some item, first it checks the cache directory to determine whether the disk page containing the item is in the DBMS cache. If it is not, cache. It may be necessary to replace (or flush) some of the cache buffers to make space available for the new item.
- The entries in the DBMS cache directory hold additional information relevant to buffer management. Associated with each buffer in the cache is a ***dirty bit***, which can be included in the directory entry to indicate whether or not the buffer has been modified. When a page is first read from the database disk into a cache buffer, a new entry is inserted in the cache directory with the new disk page address, and the dirty bit is set to 0 (zero). As soon as the buffer is modified, the dirty bit for the corresponding directory entry is set to 1 (one). Additional information, such as the transaction id(s) of the transaction(s) that modified the buffer, are also kept in the directory. When the buffer contents are replaced (flushed) from the cache, the contents must first be written back to the corresponding disk

page only if its dirty bit is 1. Another bit, called the ***pin-unpin bit***, is also needed—a page in the cache is pinned (bit value 1 (one)) if it cannot be written back to disk as yet. For example, the recovery protocol may restrict certain buffer pages from being written back to the disk until the transactions that changed this buffer have committed.

- Two main strategies can be employed when flushing a modified buffer back to disk.
  - **in-place updating** writes the buffer to the same original disk location, thus overwriting the old value of any changed data items on disk. Hence, a single copy of each database disk block is maintained.
  - **Shadowing** strategy writes an updated buffer at a different disk location, so multiple versions of data items can be maintained, but this approach is not typically used in practice.

- In general, the old value of the data item before updating is called the before image (BFIM), and the new value after updating is called the after image (AFIM). If shadowing is used, both the BFIM and the AFIM can be kept on disk; hence, it is not strictly necessary to maintain a log for recovering.

3. Write-Ahead Logging, Steal/No-Steal, and Force/No-Force:

- When in-place updating is used, it is necessary to use a log for recovery. In this case, the recovery mechanism must ensure that the BFIM of the data item is recorded in the appropriate log entry and that the log entry is flushed to disk before the BFIM is overwritten with the AFIM in the database on disk. This process is generally known as write-ahead logging and is necessary so we can UNDO the operation if this is required during recovery.
- Two types of log entry information included for a write command: the information needed for UNDO and the information needed for REDO.
  - A REDO-type log entry includes the new value (AFIM) of the item written by the operation since this is needed to redo the effect of the operation from the log (by setting the item value in the database on disk to its AFIM).
  - The UNDO-type log entries include the old value (BFIM) of the item since this is needed to undo the effect of the operation from the log (by setting the item value in the database back to its BFIM). In an UNDO/REDO algorithm, both BFIM and AFIM are recorded into a single log entry. Additionally, when cascading rollback is possible, `read_item` entries in the log are considered to be UNDO-type entries.

- The DBMS cache holds the cached database disk blocks in main memory buffers. The DBMS cache includes not only data file blocks, but also index file blocks and log file blocks from the disk.

- When a log record is written, it is stored in the current log buffer in the DBMS cache. The log is simply a sequential (append only) disk file, and the DBMS cache may contain several log blocks in main memory buffers (typically, the last n log blocks of the log file).
- When a data block stored in the DBMS cache is updated, an associated log record is written to the last log buffer in the DBMS cache. With the write-ahead logging approach, the log buffers (blocks) that contain the associated log records for a particular data block update must first be written to disk before the data block itself can be written back to disk from its main memory buffer.

- Standard DBMS recovery terminology includes the terms steal/no-steal and force/no-force, which specify the rules that govern when a page from the database cache can be written to disk:
  1. If a cache buffer page updated by a transaction cannot be written to disk before the transaction commits, the recovery method is called a ***no-steal approach***. The pin-unpin bit will be set to 1 (pin) to indicate that a cache buffer cannot be written back to disk. If the recovery protocol allows writing an updated buffer before the transaction commits, it is called steal. Steal is used when the DBMS cache (buffer) manager needs a buffer frame for another transaction and the buffer manager replaces an existing page that had been updated but whose transaction has not committed. The no-steal rule means that UNDO will never be needed during recovery, since a committed transaction will not have any of its updates on disk before it commits.
  2. If all pages updated by a transaction are immediately written to disk before the transaction commits, the recovery approach is called a ***force approach***. Otherwise, it is called ***no-force***. The force rule means that REDO will never be needed during recovery, since any committed transaction will have all its updates on disk before it is committed.

- The deferred update (NO-UNDO) recovery scheme follows a no-steal approach. However, typical database systems employ a steal/no-force (UNDO/REDO) strategy.
- Advantage of steal: It avoids the need for a very large buffer space to store all updated pages in memory.

  Advantage of no-force: An updated page of a committed transaction may still be in the buffer when another transaction needs to update it, thus eliminating the I/O cost to write that page multiple times to disk and possibly having to read it again from disk. This may provide a substantial saving in the number of disk I/O operations when a specific page is updated heavily by multiple transactions.
- To permit recovery when in-place updating is used, the appropriate entries required for recovery must be permanently recorded in the log on disk before changes are applied to the database. For example, consider the following write-ahead logging (WAL) protocol for a recovery algorithm that requires both UNDO and REDO:

1. The before image of an item cannot be overwritten by its after image in the database on disk until all UNDO-type log entries for the updating transaction up to this point have been force-written to disk.
2. The commit operation of a transaction cannot be completed until all the REDO-type and UNDO-type log records for that transaction have been force-written to disk.

- To facilitate the recovery process, the DBMS recovery subsystem may need to maintain a number of lists related to the transactions being processed in the system. These include a list for active transactions that have started but not committed as yet, and they may also include lists of all committed and aborted transactions since the last checkpoint. Maintaining these lists makes the recovery process more efficient.

4. Checkpoints in the System Log and Fuzzy Checkpointing:

- A [checkpoint, list of active transactions] record is written into the log periodically at that point when the system writes out to the database on disk all DBMS buffers that have been modified. As a consequence of this, all transactions that have their [commit, T] entries in the log before a [checkpoint] entry do not need to have their WRITE operations redone in case of a system crash, since all their updates will be recorded in the database on disk during checkpointing.
- As part of checkpointing, the list of transaction ids for active transactions at the time of the checkpoint is included in the checkpoint record, so that these transactions can be easily identified during recovery.
- The recovery manager of a DBMS must decide at what intervals to take a checkpoint. The interval may be measured in time—say, every **m** minutes—or in the number **t** of committed transactions since the last checkpoint, where the values of *m* or *t* are system parameters.
- Taking a checkpoint consists of the following actions:
    1. Suspend execution of transactions temporarily.
    2. Force-write all main memory buffers that have been modified to disk.
    3. Write a [checkpoint] record to the log, and force-write the log to disk.
    4. Resume executing transactions.

  As a consequence of step 2, a checkpoint record in the log may also include additional information, such as a list of active transaction ids, and the locations (addresses) of the first and most recent (last) records in the log for each active transaction. This can facilitate undoing transaction operations in the event that a transaction must be rolled back.

- The time needed to force-write all modified memory buffers may delay transaction processing because of step 1, which is not acceptable in practice. To overcome this, it is common to use a technique called

fuzzy checkpointing. In this technique, the system can resume transaction processing after a [begin_checkpoint] record is written to the log without having to wait for step 2 to finish. When step 2 is completed, an [end_checkpoint, ...] record is written in the log with the relevant information collected during checkpointing. However, until step 2 is completed, the previous checkpoint record should remain valid. To accomplish this, the system maintains a file on disk that contains a pointer to the valid checkpoint, which continues to point to the previous checkpoint record in the log. Once step 2 is concluded, that pointer is changed to point to the new checkpoint in the log.

5. Transaction Rollback and Cascading Rollback:

- If a transaction fails for whatever reason after updating the database, but before the transaction commits, it may be necessary to roll back the transaction. If any data item values have been changed by the transaction and written to the database on disk, they must be restored to their previous values (BFIMs).
- The undo-type log entries are used to restore the old values of data items that must be rolled back. If a transaction T is rolled back, any transaction S that has, in the interim, read the value of some data item X written by T must also be rolled back. Similarly, once S is rolled back, any transaction R that has read the value of some data item Y written by S must also be rolled back; and so on. This phenomenon is called cascading rollback, and it can occur when the recovery protocol ensures recoverable schedules but does not ensure strict or cascadeless schedules. All recovery mechanisms are designed so that cascading rollback is never required.

Figure 15 shows an example where cascading rollback is required. The read and write operations of three individual transactions are shown in Figure 15(a).

T_1
read_item(A)
read_item(D)
write_item(D)

T_2
read_item(B)
write_item(B)
read_item(D)
write_item(D)

T_3
read_item(C)
write_item(B)
read_item(A)
write_item(A)

Figure 15 (a)

Figure 15(b) shows the system log at the point of a system crash for a particular execution schedule of these transactions. The values of data items A, B, C, and D, which are used by the transactions, are shown to the right of the system log entries. Assume that the original item values, shown in the first line, are A = 30, B = 15, C = 40, and D = 20. At the point of system failure, transaction T_3 has not

reached its conclusion and must be rolled back. The WRITE operations of T_3, marked by a single * in Figure 15(b), are the T_3 operations that are undone during transaction rollback.

		A	B	C	D
		30	15	40	20
	[start_transaction, T_3]				
	[read_item, T_3, C]				
*	[write_item, T_3, B, 15, 12]		12		
	[start_transaction, T_2]				
	[read_item, T_2, B]				
**	[write_item, T_2, B, 12, 18]		18		
	[start_transaction, T_1]				
	[read_item, T_1, A]				
	[read_item, T_1, D]				
	[write_item, T_1, D, 20, 25]				25
	[read_item, T_2, D]				
**	[write_item, T_2, D, 25, 26]				26
	[read_item, T_3, A]				
	System crash				

* T_3 is rolled back because it did not reach its commit point.

** T_2 is rolled back because it reads the value of item B written by T_3.

Figure 15 (b): System log at point of crash

Figure 15(c) graphically shows the operations of the different transactions along the time axis.

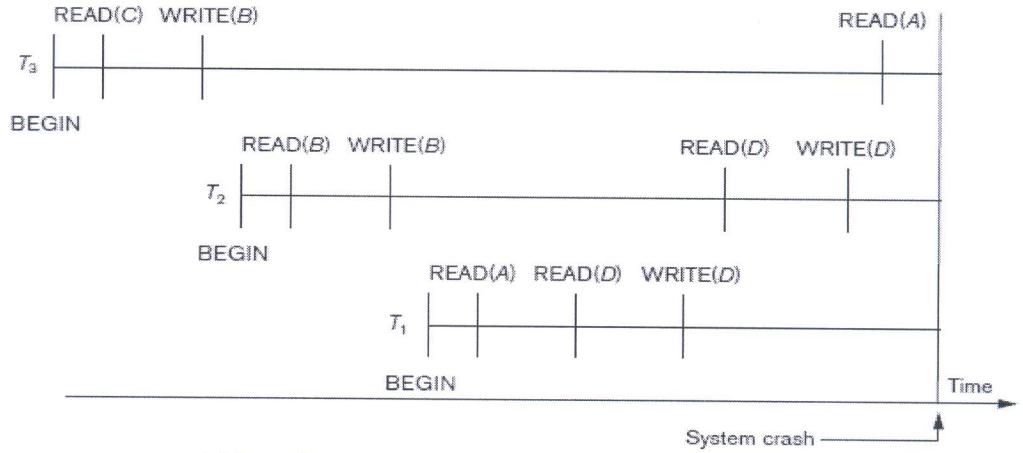

Figure 15(c): Operations before the crash

From fig 15(c), the transaction T_2 reads the value of item B that was written by transaction T_3; this can also be determined by examining the log. Because T_3 is rolled back, T_2 must now be rolled back, too. The WRITE operations of T2, marked by ** in the log, are the ones that are undone. Note that only *write_item* operations need to be undone during transaction rollback; *read_item* operations are recorded in the log only to determine whether cascading rollback of additional transactions is necessary.

In practice, cascading rollback of transactions is never required because practical recovery methods guarantee cascadeless or strict schedules. Hence, there is also no need to record any *read_item* operations in the log because these are needed only for determining cascading rollback.

6. Transaction Actions That Do Not Affect the Database:

In general, a transaction will have actions that do not affect the database, such as generating and printing messages or reports from information retrieved from the database. If a transaction fails before completion, we may not want the user to get these reports, since the transaction has failed to complete. If such erroneous reports are produced, part of the recovery process would have to inform the user that these reports are wrong, since the user may take an action that is based on these reports and that affects the database. Hence, such reports should be generated only after the transaction reaches its commit point. A common method of dealing with such actions is to issue the commands that generate the reports but keep them as batch jobs, which are executed only after the transaction reaches its commit point. If the transaction fails, the batch jobs are canceled.

5.13 NO-UNDO/REDO RECOVERY BASED ON DEFERRED UPDATE

- The idea behind deferred update is to defer or postpone any actual updates to the database on disk until the transaction completes its execution successfully and reaches its commit point.
- During transaction execution, the updates are recorded only in the log and in the cache buffers. After the transaction reaches its commit point and the log is force-written to disk, the updates are recorded in the database. If a transaction fails before reaching its commit point, there is no need to undo any operations because the transaction has not affected the database on disk in any way. Therefore, only REDO type log entries are needed in the log, which include the new value (AFIM) of the item written by a write operation. The UNDO-type log entries are not needed since no undoing of operations will be required during recovery.
- This may simplify the recovery process, but it cannot be used in practice unless transactions are short and each transaction changes few items. For other types of transactions, there is the potential for running out of buffer space because transaction changes must be held in the cache buffers until the commit point, so many cache buffers will be pinned and cannot be replaced.

- A typical deferred update protocol is stated as follows:
  1. A transaction cannot change the database on disk until it reaches its commit point; hence all buffers that have been changed by the transaction must be pinned until the transaction commits (this corresponds to a no-steal policy).

2. A transaction does not reach its commit point until all its REDO-type log entries are recorded in the log and the log buffer is force-written to disk.

The step 2 of this protocol is a restatement of the write-ahead logging (WAL) protocol. Because the database is never updated on disk until after the transaction commits, there is never a need to UNDO any operations. REDO is needed in case the system fails after a transaction commits but before all its changes are recorded in the database on disk. In this case, the transaction operations are redone from the log entries during recovery.

- For multiuser systems with concurrency control, the concurrency control and recovery processes are interrelated. Consider a system in which concurrency control uses strict two-phase locking, so the locks on written items remain in effect until the transaction reaches its commit point. After that, the locks can be released. This ensures strict and serializable schedules. Assuming that [*checkpoint*] entries are included in the log, a possible recovery algorithm for this case, called RDU_M (Recovery using Deferred Update in a Multiuser environment) is given.

Procedure RDU_M (NO-UNDO/REDO with checkpoints): Use two lists of transactions maintained by the system: the committed transactions T since the last checkpoint (commit list), and the active transactions T′ (active list). REDO all the WRITE operations of the committed transactions from the log, in the order in which they were written into the log. The transactions that are active and did not commit are effectively canceled and must be resubmitted.

The REDO procedure is defined as follows:

Procedure REDO (WRITE_OP): Redoing a write_item operation WRITE_OP consists of examining its log entry [`write_item, T, X, new_value`] and setting the value of item X in the database to new_value, which is the after image (AFIM).

Figure 16 illustrates a timeline for a possible schedule of executing transactions. Transactions T_3 and T_4 had not. Before the system crash at time t_2, T_3 and T_2 were committed but not T_4 and T_5. According to the RDU_M method, there is no need to redo the write_item operations of transaction T_1—or any transactions committed before the last checkpoint time t_1. The write_item operations of T_2 and T_3 must be redone, however, because both transactions reached their commit points after the last checkpoint. The log is force-written before committing a transaction. Transactions T_4 and T_5 are ignored: They are effectively canceled or rolled back because none of their write_item operations were recorded in the database on disk under the deferred update protocol (no-steal policy).

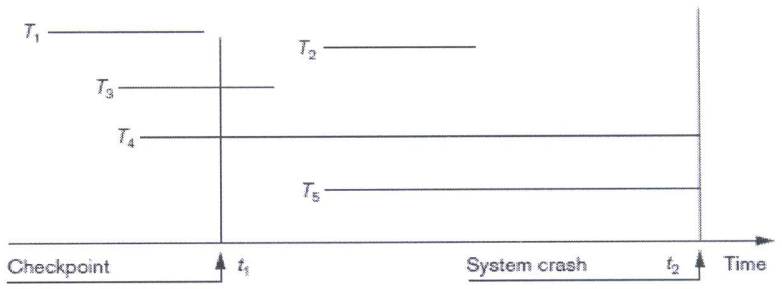

Figure 16: Example of recovery timeline to illustrate the effect of checkpointing

The NO-UNDO/REDO recovery algorithm can be made more efficient by noting that, if a data item X has been updated—as indicated in the log entries—more than once by committed transactions since the last checkpoint, it is only necessary to REDO the last update of X from the log during recovery because the other updates would be overwritten by this last REDO. In this case, start from the end of the log so that whenever an item is redone, it is added to a list of redone items. Before REDO is applied to an item, the list is checked; if the item appears on the list, it is not redone again, since its latest value has already been recovered.

If a transaction is aborted for any reason (say, by the deadlock detection method), it is simply resubmitted, since it has not changed the database on disk. A drawback of the method described here is that it limits the concurrent execution of transactions because all write-locked items remain locked until the transaction reaches its commit point. Additionally, it may require excessive buffer space to hold all updated items until the transactions commit. The method's main benefit is that transaction operations never need to be undone, for two reasons:

1. A transaction does not record any changes in the database on disk until after it reaches its commit point—that is, until it completes its execution successfully. Hence, a transaction is never rolled back because of failure during transaction execution.
2. A transaction will never read the value of an item that is written by an uncommitted transaction, because items remain locked until a transaction reaches its commit point. Hence, no cascading rollback will occur.

Figure 17 shows an example of recovery for a multiuser system that utilizes the recovery and concurrency control method described.

T_1
read_item(A)
read_item(D)
write_item(D)

T_2
read_item(B)
write_item(B)
read_item(D)
write_item(D)

T_3
read_item(A)
write_item(A)
read_item(C)
write_item(C)

T_4
read_item(B)
write_item(B)
read_item(A)
write_item(A)

(a)

(b)

[start_transaction, T_1]
[write_item, T_1, D, 20]
[commit, T_1]
[checkpoint]
[start_transaction, T_4]
[write_item, T_4, B, 15]
[write_item, T_4, A, 20]
[commit, T_4]
[start_transaction, T_2]
[write_item, T_2, B, 12]
[start_transaction, T_3]
[write_item, T_3, A, 30]
[write_item, T_2, D, 25]

← System crash

T_2 and T_3 are ignored because they did not reach their commit points.

T_4 is redone because its commit point is after the last system checkpoint.

Figure 17: Example of recovery using deferred update using concurrent execution of transactions (a) The READ and WRITE operations of four transactions. (b) System log at the point of crash.

5.14 RECOVERY TECHNIQUES BASED ON IMMEDIATE UPDATE

- In these techniques, when a transaction issues an update command, the database on disk can be updated immediately, without any need to wait for the transaction to reach its commit point.
- It is not a requirement that every update be applied immediately to disk; it is just possible that some updates are applied to disk before the transaction commits.
- Provisions must be made for undoing the effect of update operations that have been applied to the database by a failed transaction. This is accomplished by rolling back the transaction and undoing the effect of the transaction's `write_item` operations. Therefore, the UNDO-type log entries, which include the old value (BFIM) of the item, must be stored in the log. Because UNDO can be needed during recovery, these methods follow a steal strategy for deciding when updated main memory buffers can be written back to disk.
- Two main categories of immediate update algorithms.
  1. If the recovery technique ensures that all updates of a transaction are recorded in the database on disk before the transaction commits, there is never a need to REDO any operations of committed transactions. This is called the ***UNDO/NO-REDO recovery algorithm***. In this method, all updates by a transaction must be recorded on disk before the transaction commits, so that REDO is never needed. Hence, this method must utilize the steal/force strategy for deciding when updated main memory buffers are written back to disk.

2. If the transaction is allowed to commit before all its changes are written to the database, we have the most general case, known as the ***UNDO/REDO recovery algorithm***. In this case, the ***steal/no-force*** strategy is applied. This is also the most complex technique, but the most commonly used in practice.

❖ When concurrent execution is permitted, the recovery process again depends on the protocols used for concurrency control. The procedure **`RIU_M`** (Recovery using Immediate Updates for a Multiuser environment) outlines a recovery algorithm for concurrent transactions with immediate update (UNDO/REDO recovery). Assume that the log includes checkpoints and that the concurrency control protocol produces strict schedules. A strict schedule does not allow a transaction to read or write an item unless the transaction that wrote the item has committed. But deadlocks can occur in strict two-phase locking, thus requiring abort and UNDO of transactions. For a strict schedule, UNDO of an operation requires changing the item back to its old value (BFIM).

Procedure RIU_M (UNDO/REDO with checkpoints):

1. Use two lists of transactions maintained by the system: the committed transactions since the last checkpoint and the active transactions.
2. Undo all the write_item operations of the active (uncommitted) transactions, using the UNDO procedure. The operations should be undone in the reverse of the order in which they were written into the log.
3. Redo all the write_item operations of the committed transactions from the log, in the order in which they were written into the log, using the REDO procedure defined earlier.

The UNDO procedure is defined as follows:

Procedure UNDO (WRITE_OP): Undoing a `write_item` operation write_op consists of examining its log entry [`write_item, T, X, old_value, new_value`] and setting the value of item X in the database to old_value, which is the before image (BFIM). Undoing a number of write_item operations from one or more transactions from the log must proceed in the reverse order from the order in which the operations were written in the log.

Step 3 is more efficiently done by starting from the end of the log and redoing only the last update of each item X. Whenever an item is redone, it is added to a list of redone items and is not redone again. A similar procedure can be devised to improve the efficiency of step 2 so that an item can be undone at most once during recovery. In this case, the earliest UNDO is applied first by scanning the log in the

forward direction (starting from the beginning of the log). Whenever an item is undone, it is added to a list of undone items and is not undone again.

5.15 SHADOW PAGING

- This recovery scheme does not require the use of a log in a single-user environment. In a multiuser environment, a log may be needed for the concurrency control method.
- Shadow paging considers the database to be made up of a number of fixed size disk pages (or disk blocks)—say, ***n***—for recovery purposes.
- A directory with ***n*** entries is constructed, where the i^{th} entry points to the i^{th} database page on disk.
- The directory is kept in main memory if it is not too large, and all references—reads or writes—to database pages on disk go through it.
- When a transaction begins executing, the current directory—whose entries point to the most recent or current database pages on disk—is copied into a shadow directory. The shadow directory is then saved on disk while the current directory is used by the transaction.
- During transaction execution, the shadow directory is never modified. When a `write_item` operation is performed, a new copy of the modified database page is created, but the old copy of that page is not overwritten. Instead, the new page is written elsewhere—on some previously unused disk block. The current directory entry is modified to point to the new disk block, whereas the shadow directory is not modified and continues to point to the old unmodified disk block.

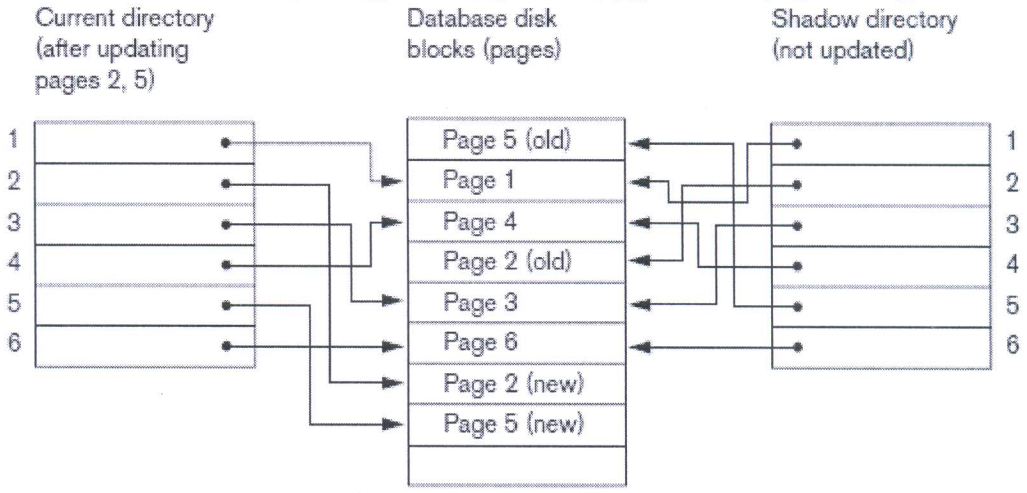

Figure 18: Example of shadow paging

Figure 18 illustrates the concepts of shadow and current directories. For pages updated by the transaction, two versions are kept. The old version is referenced by the shadow directory and the new version by the current directory.

- To recover from a failure during transaction execution, it is sufficient to free the modified database pages and to discard the current directory. The state of the database before transaction execution is available through the shadow directory, and that state is recovered by reinstating the shadow directory. The database thus is returned to its state prior to the transaction that was executing when the crash occurred, and any modified pages are discarded.
- Committing a transaction corresponds to discarding the previous shadow directory. Since recovery involves neither undoing nor redoing data items, this technique can be categorized as a NO-UNDO/NO-REDO technique for recovery.
- In a multiuser environment with concurrent transactions, logs and checkpoints must be incorporated into the shadow paging technique.

- Disadvantages of shadow paging:
  - The updated database pages change location on disk. This makes it difficult to keep related database pages close together on disk without complex storage management strategies.
  - If the directory is large, the overhead of writing shadow directories to disk as transactions commit is significant.
  - A further complication is how to handle garbage collection when a transaction commits. The old pages referenced by the shadow directory that have been updated must be released and added to a list of free pages for future use. These pages are no longer needed after the transaction commits.
  - Another issue is that the operation to migrate between current and shadow directories must be implemented as an atomic operation.

5.16 DATABASE BACKUP AND RECOVERY FROM CATASTROPHIC FAILURES

- The recovery manager of a DBMS must also be equipped to handle more catastrophic failures such as disk crashes.
- The main technique used to handle such crashes is a database backup, in which the whole database and the log are periodically copied onto a cheap storage medium such as magnetic tapes or other large capacity offline storage devices. In case of a catastrophic system failure, the latest backup copy can be reloaded from the tape to the disk, and the system can be restarted.
- Data from critical applications such as banking, insurance, stock market, and other databases is periodically backed up in its entirety and moved to physically separate safe locations.
- To avoid losing all the effects of transactions that have been executed since the last backup, it is customary to back up the system log at more frequent intervals than full database backup by periodically copying it to magnetic tape.

- The system log is usually substantially smaller than the database itself and hence can be backed up more frequently. Therefore, users do not lose all transactions they have performed since the last database backup.
- All committed transactions recorded in the portion of the system log that has been backed up to tape can have their effect on the database redone. A new log is started after each database backup. Hence, to recover from disk failure, the database is first recreated on disk from its latest backup copy on tape. Following that, the effects of all the committed transactions whose operations have been recorded in the backed-up copies of the system log are reconstructed.

QUESTION BANK:

1. How are buffering and caching techniques used by the recovery subsystem?
2. What are the before image (BFIM) and after image (AFIM) of a data item?
3. What is the difference between in-place updating and shadowing, with respect to their handling of BFIM and AFIM?
4. What are UNDO-type and REDO-type log entries?
5. Describe the write-ahead logging protocol.
6. Discuss the deferred update technique of recovery. What are the advantages and disadvantages of this technique? Why is it called the NO-UNDO/REDO method?
7. Discuss the immediate update recovery technique in both single-user and multiuser environments. What are the advantages and disadvantages of immediate update?
8. What is the difference between the UNDO/REDO and the UNDO/NO-REDO algorithms for recovery with immediate update?
9. Describe the shadow paging recovery technique. Under what circumstances does it not require a log?
10. Discuss how disaster recovery from catastrophic failures is handled.