UNIT 2 TRANSACTION MANAGEMENT AND RECOVERY

Structure **Page Nos.**

2.0 INTRODUCTION

You have already been introduced to some of the features of transactions and recovery in MCS-023, Block-2. Transactions are one of the most important components of a database management system application. A real challenge for a DBMS is not losing its integrity even when multiple transactions are going on. This unit provides a description on how concurrent transactions are handled in a DBMS. In addition, this unit also covers aspects related to recovery schemes when a number of concurrent transactions are going on. We will also explore the advanced features of DBMS in respect to Transaction Management and the recovery, transaction processing and the different methods of transaction processing. We have tried to highlight the problems that arise during concurrent transactions and the manner in which such problems may be avoided. We have explained the recovery method to ensure the consistency and ACID properties of transactions.

2.1 OBJECTIVES

After going through this unit, you should be able to:

- define transaction processing;
- describe the various protocols for concurrent transactions;
- explain the process of deadlock handling in DBMS;

- describe various recovery schemes when there is a failure, and
- define the buffer management system of the DBMS.

2.2 TRANSACTION PROCESSING

Transactions processing involves a type of computer process in which the computer responds immediately to the user's request. Each request is considered as a *transaction.* For example, each operation performed by you through the ATM at the bank is transaction.

A database system needs to have a transaction system, which ensures consistent and error free transaction execution. Transaction Systems handle errors in a safe and consistent manner, but there are certain errors that cannot be avoided (e.g., Network errors or Database deadlocks) so a method that can handle these errors must exist. It is moreover, not possible to simply abort a current process. The updates of partial transactions may be reflected in the database leaving it in an inconsistent state. This could render the entire system unusable.

That is why transaction was introduced. A Transaction runs like a sub-program that modifies the database, leading from one consistent state to another. A Transaction must be atomic (i.e., either all modifications are done, or none of them are done). Transaction Systems are designed to guarantee that Transactions are atomic.

A transaction may be defined as a collection of operations on the database that performs a single logical function in a database application or it should be an inseparable list of database operations or Logical Unit of Work (LUW). It should not violate any database integrity constraints.

When a transaction processing system creates a transaction, it will ensure that the transaction has certain characteristics. These characteristics are known as the ACID properties.

ACID is an acronym for ATOMICITY, CONSISTENCY, ISOLATION and DURABILITY.

ACID Properties

ATOMICITY: The atomicity property identifies the transaction as atomic. An atomic transaction is either fully completed, or is not begun at all.

CONSISTENCY: A transaction enforces consistency in the system state, by ensuring that at the end of the execution of a transaction, the system is in a valid state. If the transaction is completed successfully, then all changes to the system will have been made properly, and the system will be in a valid state. If any error occurs in a transaction, then any changes already made will be automatically rolled back.

ISOLATION: When a transaction runs in isolation, it appears to be the only action that the system is carrying out at a time. If there are two transactions that are performing the same function and are running at the same time, transaction isolation will ensure that each transaction thinks that it has exclusive use of the system.

DURABILITY: A transaction is durable in that once it has been successfully completed, all of the changes it has made to the system are permanent. There are safeguards that will prevent the loss of information, even in the case of system failure.

Now, let us discuss some of the traditional and advanced transaction processing methods.

Traditional Transaction Processing Methods

- **Batch Processing** is a method of computerised processing in which business transactions are accumulated over a period of time and prepared for processing as a single unit.

- **On-line Transaction Processing (OLTP)** is a method of computerised processing in which each transaction is processed immediately and the affected records are updated.

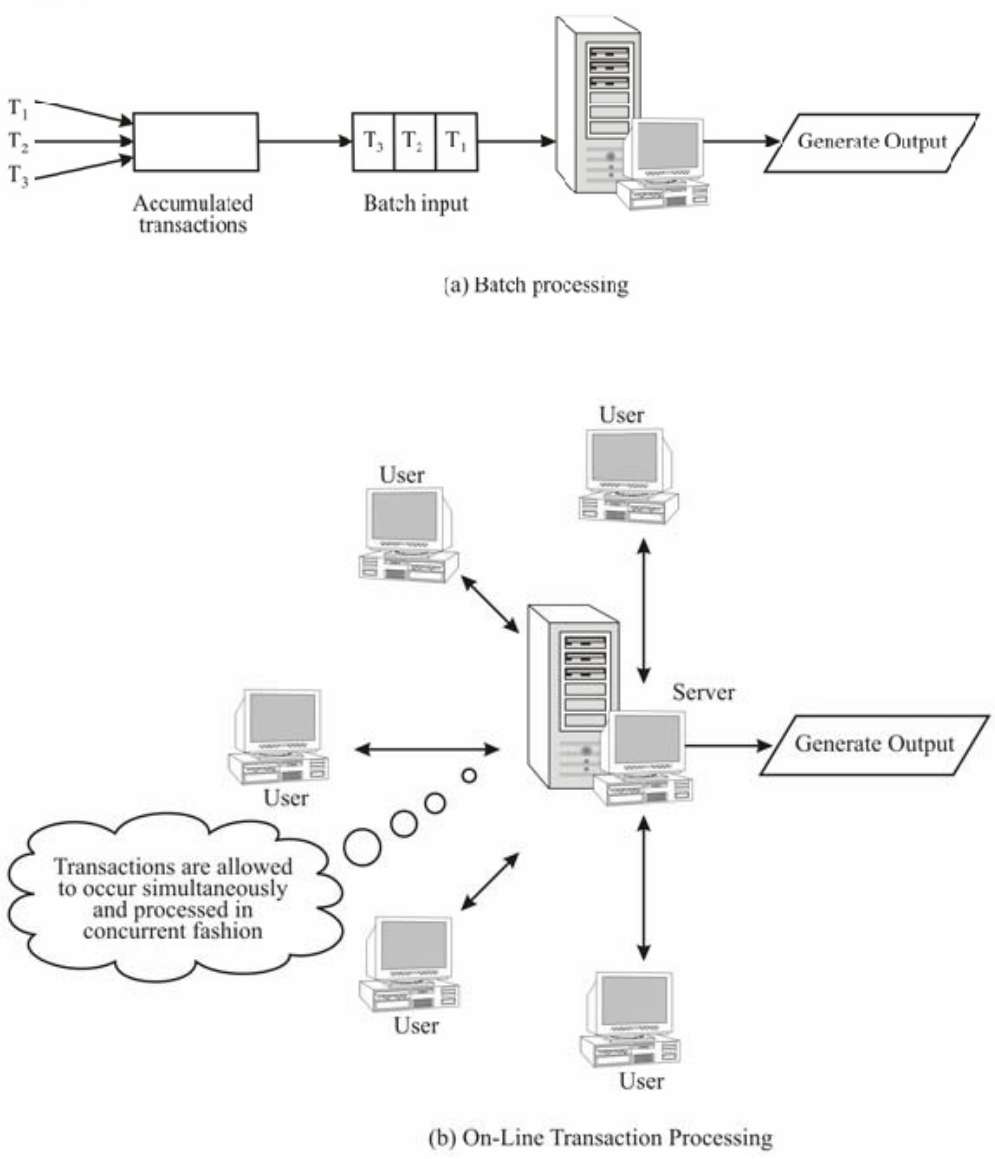


Figure 1: Batch vs. online transaction

The following are the Transaction Processing Activities:

- **Data collection:** Capturing data necessary for the transaction.
- **Data editing:** Checking validity and completeness.
  For example, a database has a value of 400 hours/week instead of 40 hours/week.
- **Data correction:** Correcting the incorrect data.
- **Data manipulation:** Calculating, summarising.
- **Data storage:** Updating the database as desired by the transaction.
- **Document production and reports:** Creating end results (paychecks).

... [2]

Advanced Transaction Processing

In contrast to the traditional transaction system there are more flexible and user oriented processing methods such as:

- Remote Backup Systems
- Transaction-Processing Monitors
- High-Performance Transaction Systems
- Real-Time Transaction Systems
- Weak Levels of Consistency (Section 2.6)
- Transactional Workflows.

Formatted: Centered

and Recovery

Comment: Please include linking para. Please note that a student has to read and understand this unit. Thus, our language should be as if we are teaching in the class. You can also ask few questions and provide information to that in text questions. Broadly the language should be conversational. Please look in the complete unit from that viewpoint

Remote Backup Systems

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

Backup site must be capable of detecting the failure, when the primary site has failed. To distinguish primary site failure from link failure it maintains several communication links between the primary and the remote backup sites.

Prior to the transfer control back to old primary when it recovers, the old primary must receive redo logs from the old backup and apply all updates locally.

Transaction Processing Monitors

TP monitors were initially developed as multithreaded servers to support large numbers of terminals from a single process. They provide infrastructure for building and administering complex transaction processing systems with a large number of clients and multiple servers.

A transaction-processing monitor has components for Input queue authorisation, output queue, network, lock manager, recovery manager, log manager, application servers, database manager and resource managers.

High-Performance Transaction Systems

High-performance hardware and parallelism helps improve the rate of transaction processing. They involve the following features for concurrency control that tries to ensure correctness without serialisability. They use database consistency constraints to split the database into sub databases on which concurrency can be managed separately.

Real-Time Transaction Systems

In systems with real-time constraints, correctness of execution involves both database consistency and the satisfaction of deadlines. Real time systems are classified as:

Hard: The task has zero value if it is completed after the deadline.
Soft : The task has diminishing value if it is completed after the deadline.

Transactional Workflows is an activity which involves the coordinated execution of multiple task performed different processing entities e.g., bank loan processing, purchase order processing.

2.3 ENHANCED LOCK BASED PROTOCOLS AND TIMESTAMP BASED PROTOCOLS

A lock based mechanism is used to maintain consistency when more than one transaction processes is executed. This mechanism controls concurrent access to data items. As per this mechanism the 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 the concurrency-control manager. The transaction can proceed only after request is granted. The Lock-compatibility matrix shows when lock request will be granted (True).

	Unlocked	**S**	**X**
S	True	True	True
X	True	False	False

The above matrix shows that if an item is locked in shared mode then only a shared lock request can be permitted. In general, 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 lock on the item, then 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. The following is an example of a transaction that requests and is granted/permitted locks:

T_2: **lock-S***(X)*;
read *(X)*;
unlock*(X)*;
lock-S*(Y)*;
read *(Y)*;
unlock*(Y)*;
display*(X+Y)*

Locking as above is not sufficient to guarantee serialisability — if *X* and *Y* are updated in-between the read of *X* and *Y* by some other transaction, the displayed sum would be wrong. A locking **protocol** is a set of rules to be followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

Pitfalls of Lock-Based Protocols

Consider the partial schedule:

T_A	T_B
lock-x(A)	
read(X)	
X=X-1000	
write (X)	lock-S (Y)
	read (Y)
	lock-S (X)
lock-X(Y)	

Neither T_A nor T_B can make progress — executing **lock-S***(X)* causes T_B to wait for T_A to release its lock on *X*, while executing **lock-X***(Y)* causes T_A to wait for T_B to release its lock on *Y*. Such a situation is called a **deadlock**. To handle a deadlock one of T_A or T_B must be rolled back and its lock released. The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil. **Starvation** may also occur, if the 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.

These concurrency control manager needs to be designed to prevent starvation.

2.3.1 Locking Protocols

They are primarily two types of looking protocols:

- Two phase locking
- Tree locking protocols.

Let us discuss them in more detail.

The Two-Phase Locking Protocols

This is a protocol, which ensures conflict-serialisable schedules read before write of transaction kindly (these transactions read a value before writing it). It consists of two phases:

Phase 1: Growing Phase

- Transaction may obtain locks
- Transaction may not release locks.

Phase 2: Shrinking Phase

- Transaction may release locks
- Transaction may not obtain locks.

The protocol assures serialisability. It can be proved that the transactions can be serialised in the order of their **lock points** (i.e., the point where a transaction acquires its final lock). Two-phase locking *does not* ensure freedom from deadlocks. Cascading roll-back is possible under two-phase locking as uncommitted values can be seen. 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 serialised in the order in which they commit.

There can be serialisable schedules that cannot be obtained if two-phase locking is used. However, in the absence of extra information (e.g., ordering of access to data), two-phase locking is needed for serialisability in the following sense:

> *Given a transaction T_i that does not follow two-phase locking, we may find another transaction T_j that uses two-phase locking, and a schedule for T_i and T_j that is not conflict serialisable.*

Lock Conversions

Sometimes shared lock may need to be upgraded to an exclusive lock. The two-phase locking with lock conversions can be defined as:

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 serialisability. 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 transaction 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. The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked.

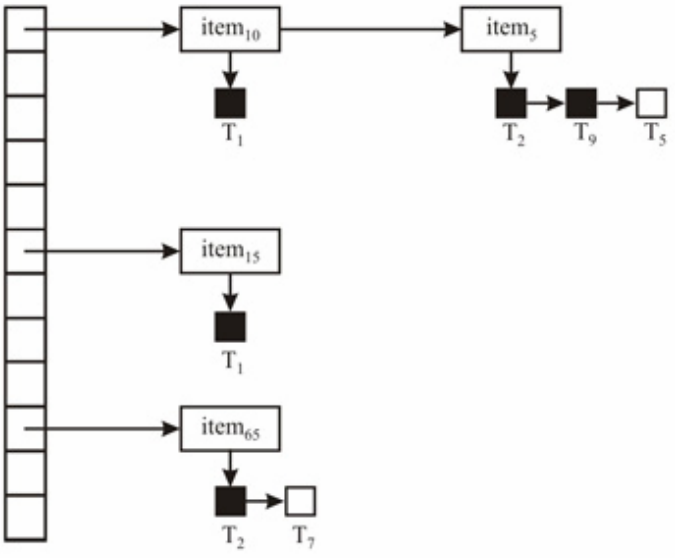


Figure 2: Lock table

... [4]

Lock Table

Black rectangles in the table indicate granted locks, white ones indicate waiting requests. A lock table also records the type of lock granted or requested. A 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. The lock manager may then keep a list of locks held by each transaction, to implement this efficiently.

Graph-Based Protocols

Graph-based protocols are an alternative to two-phase locking. They impose a partial ordering ($\rightarrow$) on the set **D** = $\{d_1, d_2, ..., d_h\}$ of all data items.

- If $d_i \rightarrow d_j$ then any transaction accessing both d_i and d_j must access d_i before accessing d_j this implies that the set **D** may now be viewed as a directed acyclic graph, called a *database graph*.

The *tree-protocol* is a simple kind of graph protocol.

Tree Protocol

In a tree based protocol only exclusive locks are permitted. The first lock by T_i may be on any data item. Subsequently, a data Q can be locked by T_i only if the parent of Q is currently locked by T_i. Data items may be unlocked at any time. *Figure 3* shows a tree based protocol for the following lock (exclusive) – unlock requests from various transactions.

Transaction T1	Transaction T2	Transaction T3
Lock A	Lock B	Lock A
Lock B	Lock D	Lock C
Lock E	Lock G	Lock F
Unlock A	Unlock B	Unlock A
Unlock B	Unlock D	Unlock C
Unlock E	Unlock G	Unlock F

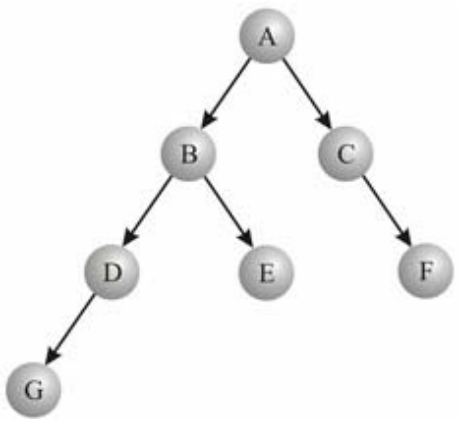


Figure 3: Tree based protocol

The tree protocol ensures conflict serialisability as well as freedom from deadlock for these transactions. Please note transaction T3 will get A only after it is set free by transaction T1. Similarly, T2 will get B when it is set free by T1. Unlocking may occur earlier in the tree-locking protocol than in the two-phase locking protocol. Thus Tree protocol is a protocol:

- with a shorter waiting time and increased concurrency,
- that is deadlock-free and does not require rollbacks,
- where aborting a transaction can still lead to cascading rollbacks.

However, in the tree-locking protocol, a transaction may have to lock data items that it does not access. Thus, it has:

- increased locking overhead, and additional waiting time.
- potential decrease in concurrency.

Schedules that are not possible in two-phase locking are possible under tree based protocol and vice versa.

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 the 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 (e.g., find all accounts in Bombay) and a transaction that inserts a tuple in the relation (e.g., insert a new account at Bombay) may be in conflict with each other, despite the fact that the two transactions are not accessing any tuple in common.
- If only tuple locks are used, non-serialisable schedules may be the result: the scan than transaction may not see the new account, yet may be serialised before the insert transaction.

The transaction scanning the relation reads the 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 to such a problem may be:

- Associating a data item with the relation helps represent the information about the tuples the relation contains.
- Transactions scanning the relation acquires a shared lock in the data item.
- Transactions inserting or deleting a tuple acquires an exclusive lock on the data item. (Note: locks on the data item do not conflict with locks on individual tuples).

The above mentioned 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: Every relation must have at least one index. Access to a relation must be made only through one of the indices on the relation. A transaction T_i that performs a lookup must lock all the index buckets that it accesses, in S-mode. A transaction T_i may not insert a tuple t_i into a relation r without updating all indices to r. T_i must perform a lookup on every index to find all index buckets that could have possibly contained a pointer to tuple t_i, had it existed already, and obtain locks in X-mode on all these index buckets. T_i must also obtain locks in X-mode on all index buckets that it modifies. The rules of the two-phase locking protocol must be observed for index locking protocols to be effective.

2.3.2 Timestamp-Based Protocols

Each transaction is issued a timestamp when it enters the system. If an old transaction T_i has time-stamp TS(T_i), a new transaction T_j is assigned time-stamp TS(T_j) such that TS(T_i) <TS(T_j). This protocol manages concurrent execution in such a manner that the time-stamps determine the serialisability order. In order to assure such behaviour, the protocol needs to maintain for each data Q two timestamp values:

... [5]

- **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 executes any conflicting **read** and **write** operations in timestamp order. Suppose a transaction T_i issues a **read**(Q).

1) If TS(T_i) <**W**-timestamp(Q), $\Rightarrow$ T_i needs to read a value of Q that was already overwritten. Action: reject **read** operation and rolled back T_i.

2) If TS(T_i)≥ **W**-timestamp(Q), $\Rightarrow$ It is OK. Action: Execute **read** operation and set R-timestamp(Q) to the maximum of R-timestamp(Q) and TS(T_i).

Suppose that transaction T_i issues **write** (Q):

1) If TS(T_i) < R-timestamp(Q), $\Rightarrow$ the value of Q that T_i is writing was used previously, this value should have never been produced. Action: Reject the **write** operation. Roll back T_i
2) If TS(T_i) < W-timestamp(Q), $\Rightarrow$ T_i is trying to write an obsolete value of Q. Action: Reject **write** operation, and roll back T_i.
3) Otherwise, execute **write** operation, and set W-timestamp(Q) to TS(T_i).

Correctness of Timestamp-Ordering Protocol

The timestamp-ordering protocol guarantees serialisability if all the arcs (please refer to MCS-023, Block-2 for rules on drawing precedence graph arcs) in the precedence graph of the form as shown in *Figure 4* and there is no cycle.

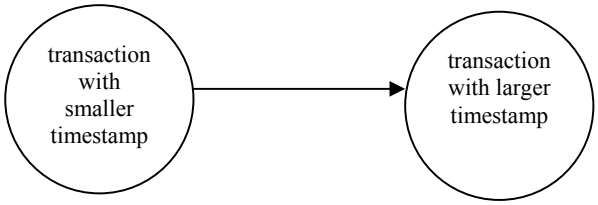


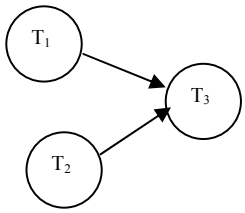
Figure 4: Arcs of a precedence graph in timestamp ordering protocol

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

Example: Use of the timestamp based Protocol is shown in the example below. Assume the following table showing a partial schedule for several data items for transactions with timestamps 1, 2, 3. Please note that all the transactions T_1 and T_2 allowed the read values of X and Y which are written by a transaction with a lower timestamp than that of these transactions. The transaction T_3 will also be allowed to write the values of X and Y as the read timestamp value of these variables is 2 and since it reads X and Y itself so the read timestamp becomes 3 for X and Y. Thus, no later transaction have the read values and hence T_3 will be allowed to write the values of X and Y.

T_1	T_2	T_3
		read (X)
	read (Y)	
read (Y)		
		read (Y)
	read (X)	
	abort	
read (X)		
		write (X)
		write (Y)

The precedence graph of these transactions would be (Please recollect the rules of drawing precedence graph).



Thus, the partial ordering of execution is T_1, T_2, T_3, or T_2, T_1, T_3.

Recoverability and Cascade Freedom

Problem with timestamp-ordering protocol:

- Suppose T_i aborts, but T_j has read a data item written by *Ti* then T_j must also be aborted; if T_j had committed earlier, the schedule is **not recoverable.**
- Also, any transaction that has read a data item written by T_j must abort too.

Thus, there may occur a cascading rollback – that is, a chain of rollbacks.

Solution:

- A transaction is structured in such a manner that all the writes are performed at the end of its processing.
- All such writes of a transaction form an atomic action; no transaction may execute while a transaction is being written.
- A transaction that aborts or is rolled back is restarted with a new timestamp.

Thomas' Write Rule: Modified versions of the timestamp-ordering protocol in which obsolete write operations may be ignored under certain circumstances. When T_i attempts to write data item Q, if TS (T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of $\{Q\}$. Hence, rather than rolling back T_i 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.

Validation-Based Protocol

In certain cases when the write operations are minimum then a simple protocol may be used. In such a protocol an execution of transaction T_i is done in three phases.

1) **Read and execution phase**: Transaction T_i writes only to temporary local variables.

2) **Validation phase**: Transaction T_i performs a "validation test" to determine whether local variables can be written without violating serialisability.

3) **Write phase**: If T_i is validated, the updates are applied to the database otherwise T_i 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. This protocol is also known as **Optimistic Concurrency Control** since the transaction executes fully in the hope that all will go well during the validation.

Each transaction T_i has 3 timestamps:

- **Start**(T_i) : The time when T_i starts the execution.
- **Validation**(T_i): The time when T_i started with the validation phase.
- **Finish**(T_i) : The time when T_i completed its write phase.

Serialisability order is determined by a timestamp given at the time of validation, to increase concurrency. Thus $TS(T_i)$ is given the value of **Validation**(T_i).

... [6]

This protocol is useful and gives a greater degree of concurrency provided the probability of conflicts is low. That is because the serialisability order is not pre-decided and relatively less transactions will have to be rolled back.

Formatted: Centered

Validation Test for Transaction T_j

If for all T_i with TS (T_i) < TS (T_j) either one of the following condition holds:

- **finish**(T_i) < **start**(T_j)
- **start**(T_j) < **finish**(T_i) < **validation**(T_j) **and** the set of data items written by T_i does not intersect with the set of data items read by T_j, then validation succeeds and T_j can be committed. Otherwise, validation fails and T_j is aborted.

Justification: If the first condition is satisfied, and there is no execution overlapping, or if the second condition is satisfied and,

- the writes of T_j do not affect reads of T_i since they occur after T_i has finished its reads.
- the writes of T_i do not affect reads of T_j since T_j does not read any item written by T_i.

Schedule Produced by Validation

Example of schedule produced using validation:

T_A	T_B: Transfer of Rs.1000/- From account X to account Y
read (X)	read (X) X=X−1000 read (Y) Y=Y+1000
	(validate) write (X) write (Y)
read (Y) (validate) The validation here may fail although no error has occurred.	

Formatted: French France

2.3.3 Multiple Granularity

In the concurrency control schemes, we used each individual data item as a unit on which synchronisation is performed. Can we allow data items to be of various sizes and define a hierarchy of data granularities, whereby the small granularities are nested within larger ones? It can be represented graphically as a tree (not tree-locking protocol). In such a situation, when a transaction locks a node in the tree explicitly, it implicitly locks all the node's descendents in the same mode. Granularity of locking (the levels in tree where locking is done) are:

- fine granularity (lower in the tree): It allows high concurrency but has high locking overhead,
- coarse granularity (higher in the tree): It has low locking overhead but also has low concurrency.

Figure 5 shows an example of Granularity Hierarchy.

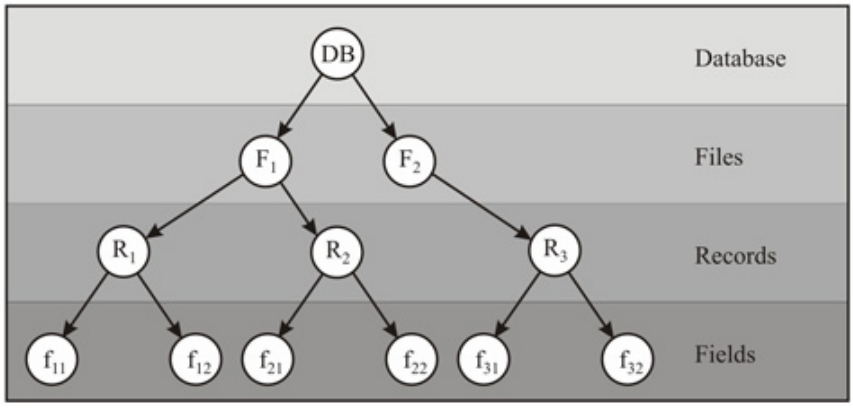


Figure 5: Hierarchical locking

The highest level in the example hierarchy is the entire database. The levels below are of file, record and field.

But how is locking done in such situations? Locking may be done by using intention mode locking.

Intention Lock Modes

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

- ***Intention-shared*** (IS): indicates explicit locking at a lower level of the tree but only with shared locks.
- ***Intention-exclusive*** (IX): indicates explicit locking at a lower level with exclusive or shared locks
- ***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 share (S) or exclusive (X) mode without having to check all descendent nodes.

Thus, this locking scheme helps in providing more concurrency but lowers the lock overheads.

Compatibility Matrix with Intention Lock Modes: The following figure shows the compatible matrix that allows locking.

	IS	IX	S	S IX	X
IS	✓	✓	✓	✓	×
IX	✓	✓		×	×
X	✓	×	✓	×	×
S IX	✓	×	×	×	×
X	×	×	×	×	×

☞ Check Your Progress 1

1) How do you define transaction processing methods? Also list the advanced transaction methods.

……

……

……

2) Why we use lock based protocol? What are the pitfalls of lock based protocol?

……

……

……

3) What is a timestamp? What is the need of time stamping protocol?

……

……

……

4) How are insert and delete operations performed in a tree protocol?

……

……

……

5) Define multiple granularity?

……

……

……

6) Test all the protocols discussed till now.

……

……

……

Formatted: Centered

and Recovery

2.4 MULTI-VERSION SCHEMES

Multi-version schemes maintain old versions of data item to increase concurrency. These schemes are available for:

- Multi version Timestamp Ordering
- Multi-version Two-Phase Locking.

In multi-version schemes each successful **write** results in the creation of a new version of the data item written.

Timestamps can be used to label the versions. When a **read** (Q) operation is issued, you can select an appropriate version of Q based on the timestamp of the transaction, and return the value of the selected version. **Read** operation never has to wait as an appropriate version is returned immediately.

2.4.1 Multi-Version Timestamp Ordering

Each data item Q has a sequence of versions <Q_1, Q_2,...., Q_m>. Each version Q_k contains three data fields:

Content−the value of version Q_k.
W-timestamp (Q_k) − timestamp of the transaction that created (wrote) the version Q_k
R-timestamp (Q_k) − largest timestamp of a transaction that successfully read version Q_k.

When a transaction T_i creates a new version Q_k of Q, Q_k's W-timestamp and R-timestamp are initialised to TS(T_i). R-timestamp of Q_k is updated whenever a transaction T_j reads Q_k, and TS(T_j) > R-timestamp(Q_k).

1) If transaction Ti issues a read(Q) , then the value returned is the content of version Qk.

2) If transaction Ti issues a write(Q), and if TS(Ti) < R-timestamp(Qk), then Transaction Ti is rolled back, otherwise, if TS(Ti)= W- timestamp(Qk), the contents of Qk are overwritten a new version of Q is created.

The following multi-version technique ensures serialisability.

Suppose that transaction T_i issues a read (Q) or write(Q) operation. Let Q_k denote the version of Q whose write timestamp is the largest write timestamp less than or equal to TS(T_i).

1) If transaction T_i issues a read (Q), then the value returned is the content of version Q_k.
2) If transaction T_i issues a write (Q), and if TS(T_i) < R-timestamp(Q_k), then transaction T_i is rolled back. Otherwise, if TS(T_i) = W-timestamp(Q_k), the contents of Q_k are overwritten, then a new version of Q is created.

Read always succeeds. A write by T_i is rejected if some other transaction T_j (in the serialisation order defined by the timestamp values) should read T_i's write, has already read a version created by a transaction older than T_i.

2.4.2 Multi-Version Two-Phase Locking

It 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.

1) Each successful **write** results in the creation of a new version of the data item written.
2) Each version of a data item has a single timestamp whose value is obtained from a counter-transaction that is incremented during commit processing.

Read-only transactions are assigned a timestamp. By reading the current value of transaction counter before they start execution; they follow the multi-version timestamp-ordering protocol for performing the read function.

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 T_i completes and commit processing occurs:

1) T_i sets timestamp on the versions it has created to transaction-counter + 1
2) T_i increments transaction-counter by 1.

Read-only transactions that start after T_i increments transaction-counter will see the values updated by T_i. Read-only transactions that start before T_i increments the transaction counter, will see the value before the updates by T_i. Only serialisable schedules are produced using this method.

2.5 DEADLOCK HANDLING

Consider the following two transactions:

T_1:	write (X)	T_2:	write(Y)
	write(Y)		write(X)

A schedule with deadlock is given as:

T 1	T 2
lock-X on X	
write (X)	
	lock-X on Y
	write (X)
	wait for **lock-X** on X
wait for **lock-X** on Y	

A system is deadlocked if a set of transactions in the set is waiting for another transaction in the set. Let us discuss certain mechanisms to deal with this deadlock.

2.5.1 Deadlock Prevention

Deadlock prevention protocols ensure that the system will never enter into a deadlock state. The basic prevention strategies are:

- The strategies require that each transaction lock all its data items before it begins execution (pre declaration).
- They 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).

Two common techniques for deadlock prevention are *wait-die* and *wound-wait.* In both the *wait-die* and *wound-wait* schemes, a rolled back transaction is restarted with its original timestamp. Older transactions thus have precedence over newer ones, and deadlock is hence avoided.

These schemes use transaction timestamps for the prevention of deadlock.

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 the needed data item

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.

Timeout-Based Schemes:

The timeout-based schemes have the following characteristics:

1) A transaction waits for a lock only for a specified amount of time. After that, the wait times out and transaction are rolled back. Thus deadlocks are prevented.

2) Simple to implement; but starvation may occur. Also it is difficult to determine the good value of timeout interval.

2.5.2 Deadlock Detection

Deadlocks can be detected using 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 $T_i \rightarrow T_j$.

If $T_i \rightarrow T_j$ is in E, then there is a directed edge from T_i to T_j, implying that T_i is waiting for T_j to release a data item. When T_i requests a data item currently being held by T_j, then the edge $T_i \rightarrow T_j$ is inserted in the wait-for graph. This edge is removed only when T_j is no longer holding a data item needed by T_i.

The system is in a state of deadlock if the wait-for graph has a cycle. You must make use of a deadlock-detection algorithm periodically to look for such cycles.

Figure 6: Deadlock detection

2.5.3 Deadlock Recovery

When a deadlock is detected, some transaction will have to be rolled back to break the deadlock. Selecting that transaction, as a victim will incur minimum cost. Rollback will determine the distance the transaction needs to be rolled back. Total rollback aborts the transaction and then restarts it. The more effective method of deadlock recovery is to rollback transaction only as far as necessary to break the deadlock. Starvation occurs if the same transaction is always selected as a victim. You can include the number of rollbacks in the cost factor calculation in order to avoid starvation.

2.6 WEAK LEVELS OF CONSISTENCY

The protocols such as strict two phase locking protocol restricts concurrency while transactions are being execution. Can we allow more concurrency by compromising on correctness/accurateness, which now needs to be ensured by database programmers rather than by the DBMS? We can operate on weak levels of consistency using the following mechanism:

Degree-two 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, however,

- X-locks must be held till the transaction has ended
- Serialisability is not guaranteed. The programmer must ensure that no erroneous database state will occur.

One of the degree two-consistency level protocols is Cursor stability. It has the following rules:

- For reads, each tuple is locked, read, and lock is immediately released
- X-locks are held till end of transaction.

Weak Levels of Consistency in SQL

SQL allows non-serialisable executions.

- **Serialisable** 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) that is, 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. This is the level, which has almost no restriction on concurrency but will result in all sorts of concurrency related problems.

2.7 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 leads to low concurrency. Two-phase locking on an index may result in transactions being executed practically one-at-a-time. It is acceptable to have nonserialisable 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. There are index concurrency protocols where locks on internal nodes are released early, and not in a two-phase fashion.

... [9]

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 the parent in exclusive mode.

2.8 FAILURE CLASSIFICATION

A DBMS may encounter a failure. These failure may be of the following types:
Transaction failure. An ongoing transaction may fail due to:

- **Logical errors**: Transaction cannot be completed due to some internal error condition.
- **System errors**: The database system must terminate an active transaction due to an error condition (e.g., deadlock).

System crash: A power failure or other hardware or software failure causes the system to crash.

- **Fail-stop assumption**: Non-volatile storage contents are assumed to be uncorrupted by system crash.
- **Disk failure**: A head crash or similar disk failure destroys all or part of the disk storage capacity.
- **Destruction is assumed to be detectable**: Disk drives use checksums to detect failure.

All these failures result in the inconsistent state of a transaction. Thus, we need a recovery scheme in a database system, but before we discuss recovery. Let us briefly define the storage structure from a recovery point of view.

Storage Structure

There are various ways for storing information.

Volatile storage

- does not survive system crashes
- examples: main memory, cache memory

Nonvolatile storage

- survives system crashes
- examples: disk, tape, flash memory, non-volatile (battery backed up) RAM.

Stable storage

- a mythical form of storage that survives all failures
- approximated by maintaining multiple copies on distinct nonvolatile media.

Stable-Storage Implementation

A stable storage maintains multiple copies of each block on separate disks. Copies can be kept at remote sites to protect against disasters such as fire or flooding. Failure during data transfer can still result in inconsistent copies. A block transfer can result in:

- Successful completion
- Partial failure: destination block has incorrect information
- Total failure: destination block was never updated.

For protecting storage media from failure during data transfer we can execute output operation as follows (assuming two copies of each block):

1) Write the information on the first physical block.
2) When the first write successfully completes, write the same information on the second physical block.
3) The output is completed only after the second write is successfully completed.

Copies of a block may differ due to failure during output operation. To recover from this failure you need to first find the inconsistent blocks:

One Expensive solution: is comparing the two copies of every disk block.

A Better solution may be to:

- record in-progress disk writes on non-volatile storage (Non-volatile RAM or special area of disk),
- use this information during recovery to find blocks that may be inconsistent, and only compare copies of these, and
- used in hardware-RAID systems.

If either copy of an inconsistent block is detected with an error (bad checksum), overwrite it with the other copy. If both have no error, but are different, overwrite the second block with the first block.

☞ Check Your Progress 2

1) Define multi-version scheme?

……

……

2) Define dead locks? How can dead locks be avoided?

……

……

3) Define weak level of consistency in SQL?

……

……

4) How is stable storage implemented?

……

……

2.9 RECOVERY ALGORITHMS

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

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

While modifying the database, without ensuring that the transaction will commit, may leave the database in an inconsistent state. Let us consider an example transaction T_1 that transfers Rs.1000/- from account X to account Y; goal is either to perform all database modifications made by T_1 or none at all. T_1 modifies x by subtracting Rs.1000/- and modifies Y by adding Rs.1000/-. A failure may occur after one of these modifications has been made, but before all of them are made. To ensure consistency despite failures, we have several recovery mechanisms.

Let us now explain two approaches: **log-based recovery** and **shadow paging**.

Formatted: Centered

and Recovery

2.9.1 Log-Based Recovery

A log is maintained on a stable storage media. The log is a sequence of **log records**, and maintains a record of update activities on the database. When transaction T_i starts, it registers itself by writing a

$<T_i$ **start**>log record

Before T_i executes **write**(X), a log record $<T_i, X, V_1, V_2>$ is written, where V_1 is the value of X before the write (undo value), and V_2 is the value to be written to X *(redo value).*

- Log record notes that T_i has performed a write on data item ***X***. ***X*** had value V_1 before the write, and will have value V_2 after the write.

- When T_i finishes it last statement, the log record **<**T_i **commit>** is written. We assume for now that log records are written directly to a stable storage media (that is, they are not buffered).

Two approaches for recovery using logs are:

- Deferred database modification.
- Immediate database modification.

Deferred Database Modification

The **deferred database modification** scheme records all the modifications to the log, but defers all the **write**s to after partial commit. Let us assume that transactions execute serially, to simplify the discussion.

A transaction starts by writing $<T_i$ ***start***> record to log. A **write**(X) operation results in a log record $<T_i, X, V>$ being written, where V is the new value for X. The write is not performed on X at this time, but is deferred. When T_i partially commits, $<T_i$ **commit**> is written to the log. Finally, the log records are read and used to actually execute the previously deferred writes. During recovery after a crash, a transaction needs to be redone if both $<T_i$ **start**> and$<T_i$ **commit**> are there in the log. Redoing a transaction T_i (**redo**T_i) sets the value of all data items updated by the transaction to the new values. Crashes can occur while

- the transaction is executing the original updates, or
- while recovery action is being taken.

Example:

Transactions T_1 and T_2 (T_1 executes before T_2):

T_1: **read** (X)	T_2 : **read** (Z)
$X= X–1000$	Z= $Z– 1000$
Write (X)	**write** (Z)
read (Y)	
$Y= Y + 1000$	
write (Y)	

Formatted: French France

The following figure shows the log as it appears at three instances of time (Assuming that initial balance in X is 10,000/- Y is 8,000/- and Z has 20,000/-):

(a)	(b)	(c)
<T_1 start>	<T_1 start>	<T_1 start>
<T_1, X 9000>	<T_1, X 9000>	<T_1, X 9000>
<T_1, Y 9000>	<T_1, Y 9000>	<T_1, Y 9000>
	<T_1, Commit>	<T_1, Commit>
	<T_2 start>	<T_2 start>
	<T_2, Z 19000>	<T_2, Z 19000>
		<T_2, Commit>

Formatted: French France

If log on stable storage at the time of crash as per (a) (b) and (c) then in:

(a) No redo action needs to be performed.

(b) redo(T_1) must be performed since <T_1 **commit**> is present

(c) redo(T_2) must be performed followed by redo(T_2) since

<T_1 **commit**> and <T_2 **commit**> are present.

Please note that you can repeat this sequence of redo operation as suggested in (c) any number of times, it will still bring the value of X, Y, Z to consistent redo values. This property of the redo operation is called **idempotent.**

Immediate Database Modification

The **immediate database modification** scheme allows database updates on the stored database even of an uncommitted transaction. These updates are made as the writes are issued (since undoing may be needed, update logs must have both the **old value** as well as the **new value**). Updated log records must be written *before* database item is written (assume that the log record is output directly to a stable storage and can be extended to postpone log record output, as long as prior to execution of an **output** (Y) operation for a data block Y all log records corresponding to items Y 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.

Example:

Log	**Write operation**	**Output**
<T_1 **start**>		
<T_1, X, 10000, 9000>		
T_1, Y, 8000, 9000		
	$X = 9000$	Output Block of X
	$Y = 9000$	Output Block of Y
<T_1 **commit**>		
<T_2 **start**>		
<T_2, Z, 20,000, 19,000>		
	$Z = 19000$	Output Block of Z
<T_2 **commit**>		

The recovery procedure in such has two operations instead of one:

- **undo**(T_i) restores the value of all data items updated by T_i to their old values, moving backwards from the last log record for T_i,
- **redo**(T_i) sets the value of all data items updated by T_i to the new values, moving forward from the first log record for T_i.

Both operations are **idempotent, *that is*,** even if the operation is executed multiple times the effect is the same as it is executed once. (This is necessary because operations may have to be re-executed during recovery).

When recovering after failure:

- Transaction T_i needs to be undone if the log contains the record <T_i **start**>, but does not contain the record <T_i **commit**>.

- Transaction T_i needs to be redone if the log contains both the record <T_i **start**> and the record <T_i **commit**>.

Undo operations are performed first, then redo operations.

Example:

Consider the log as it appears at three instances of time.

(a)	(b)	(c)
<T_1 start>	<T_1 start>	<T_1 start>
<T_1, X 10000, 9000>	<T_1, X 10000, 9000>	<T_1, X 10000, 9000>
<T_1, Y 8000, 9000>	<T_1, Y 8000, 9000>	<T_1, Y 8000, 9000>
	<T_1, Commit>	<T_1, Commit>
	<T_2 start>	<T_2 start>
	<T_2, Z 20000, 19000>	<T_2, Z 20000, 19000>
		<T_2, Commit>

Formatted: Centered

and Recovery

Formatted: French France

Recovery actions in each case above are:

(a) undo (T_1): Y is restored to 8000 and X to 10000.

(b) undo (T_2) and redo (T_1): Z is restored to 20000, and then X and Y are set to 9000 and 9000 respectively.

(c) redo (T_1) and redo (T_2): X and Y are set to 9000 and 9000 respectively. Then *Z* is set to 19000

Checkpoints

The following problem occurs during recovery procedure:

- searching the entire log is time-consuming as we are not aware of the consistency of the database after restart. Thus, we might unnecessarily redo transactions, which have already output their updates to the database.

Thus, we can streamline recovery procedure by periodically performing **check pointing**. Check pointing involves:

- Output of all the log records currently residing in the non-volatile memory onto stable storage.
- Output all modified buffer blocks to the disk.
- Write a log record < **checkpoint**> on a stable storage.

During recovery we need to consider only the most recent transactions that started before the checkpoint and is not completed till, checkpoint and transactions started after check point. Scan backwards from end of log to find the most recent <**checkpoint**> record. Continue scanning backwards till a record <T_i **start**> is found. Need only consider part of the log following above **star**t record. The earlier part of the log may be ignored during recovery, and can be erased whenever desired. For all transactions (starting from T_i or later) with no <T_i **commit**>, execute **undo** ***(****T_i*).* (Done only in case immediate modification scheme is used). Scanning forward in the log, for all transactions starting from T_i or later with a <T_i **commit**>, execute **redo***(*T_i*).*

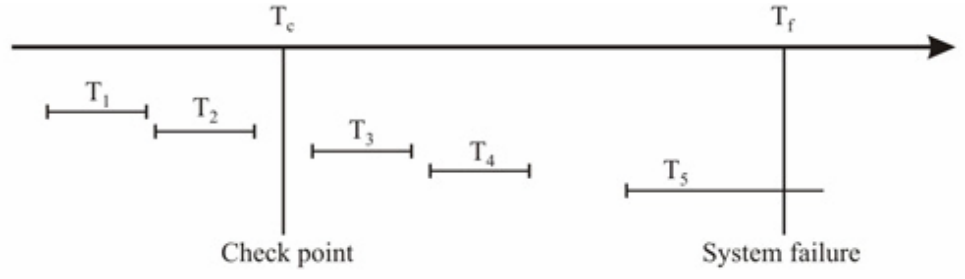


Figure 7: Checkpoint and failure

- T_1 and T_2 can be ignored (updates already output to disk due to checkpoint)
- T_3 and T_4 are redone.
- T_5 is undone.

2.9.2 Shadow Paging

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

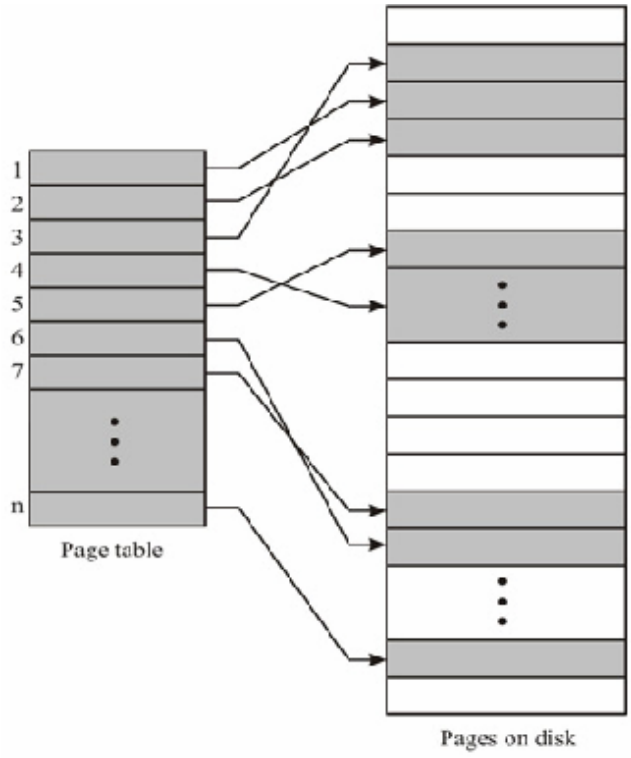


Figure 8: A Sample page table

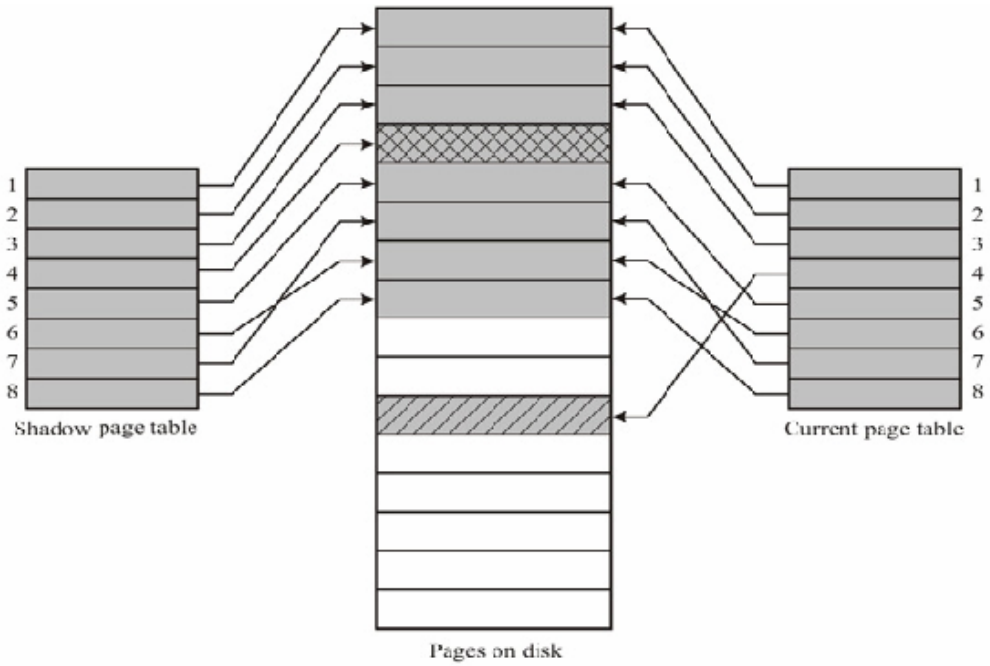


Figure 9: Shadow page table

... [14]

To commit a transaction:

1) Flush all modified pages in main memory to disk
2) Output current page table to disk
3) Make the current page table the new shadow page table, as follows:

- keep a pointer to the shadow page table at a fixed (known) location on disk.
- to make the current page table the new shadow page table, simply update the pointer to point at the current page table on disk.

Once pointer to shadow page table has been written, transaction is committed. No recovery is needed after a crash — new transactions can start right away, using the shadow page table. Pages not pointed to from current/shadow page table should be freed (garbage collected).
Advantages of shadow-paging over log-based schemes:

- It has no overhead of writing log records,
- The recovery is trivial.

Disadvantages:

- Copying the entire page table is very expensive, it can be reduced by using a page table structured like a B^+-tree (no need to copy entire tree, only need to copy paths in the tree that lead to updated leaf nodes).
- Commit overhead is high even with the above extension (Need to flush every updated page, and page table).
- Data gets fragmented (related pages get separated on disk).
- After every transaction is completed, the database pages containing old versions is completed, of modified data need to be garbage collected/freed.
- Hard to extend algorithm to allow transactions to run concurrently (easier to extend log based schemes).

2.9.3 Recovery with Concurrent Transactions

We can modify 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; logging is done as described earlier. The checkpointing technique and actions taken on recovery have to be changed since several transactions may be active when a checkpoint is performed.

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:

Initialises *undo-list* and *redo-list* to empty
Scans the log backwards from the end, stopping when the first <**checkpoint** *L*> record is found.

For each log record found during the backward scan:

- If the record contains <T_i commit>, add T_i to *redo-list.*
- If the record contains <T_i start>, then if T_i is not in *redo-list*, add T_i to *undo-list*
- For every T_i in *L*, if T_i is not in *redo-list*, add T_i 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 <T_i **start**> records have been encountered for every T_i 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.*

SQL does not have very specific commands for recovery but, it allows explicit COMMIT, ROLLBACK and other related commands.

2.10 BUFFER MANAGEMENT

When the database is updated, a lot of records are changed in the buffers allocated to the log records, and database records. Although buffer management is the job of the operating system, however, some times the DBMS prefer buffer management policies of their own. Let us discuss in details buffer management in the subsequent sub-sections.

2.10.1 Log Record Buffering

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

The rules below must be followed if log records are buffered:

- Log records are output to stable storage in the order in which they are created.
- Transaction T_i enters the commit state only when the log record <T_i **commit**> has been output to stable storage.
- Before a block of data in the main memory is output to the database, all log records pertaining to data in that block must be output to a stable storage.

These rules are also called the **write-ahead logging** scheme.

2.10.2 Database Buffering

The database maintains an in-memory buffer of data blocks, when a new block is needed, if the buffer is full, an existing block needs to be removed from the buffer. If the block chosen for removal has been updated, even then it must be output to the disk. However, as per write-ahead logging scheme, a block with uncommitted updates is output to disk, log records with undo information for the updates must be output to the log on a stable storage. No updates should be in progress on a block when it is output to disk. This can be ensured as follows:

- Before writing a data item, the 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).

A database buffer can be implemented either, in an area of real main-memory reserved for the database, or in the virtual memory. Implementing buffer in reserved main-memory has drawbacks. Memory is partitioned before-hand between database buffer and applications, thereby, limiting flexibility. Although the operating system knows how memory should be divided at any time, it cannot change the partitioning of memory.

Database buffers are generally implemented in virtual memory in spite of drawbacks. When an operating system needs to evict a page that has been modified, to make space for another page, the page is written to swap space on disk. When the database

... [15]

decides to write buffer page to disk, buffer page may be in swap space, and may have to be read from swap space on disk and output to the database on disk, resulting in extra I/O, Known as **dual paging** problem. Ideally when swapping out a database buffer page, the operating system should handover the control to the database, which in turn outputs page to database instead of to swap space (making sure to output log records first) dual paging can thus be avoided, but common operating systems do not support such functionality.

Formatted: Centered

2.11 ADVANCED RECOVERY TECHNIQUES

Advanced recovery techniques support high-concurrency locking techniques, such as those used for B^+-tree concurrency control. 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/deletions are undone by executing a deletion/insertion operation (known as **logical undo**).

For such operations, undo log records should contain the undo operation to be executed called **logical undo logging**, in contrast to **physical undo logging**. Redo information is logged **physically** (that is, new value for each write). Even for such operations logical redo is very complicated since the databases state on disk may not be "operation consistent".

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 being executed, normal log records with old and new value information are logged.
3) When operation is completed, <T_i, O_j, **operation-end**, U> is logged, where U contains information needed to perform a logical undo information.

If crash/rollback occurs before the operation is completed, 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 is completed, the **operation-end** log record is found, and in this case logical undo is performed using U; the physical undo information for the operation is ignored. Redo of operation (after crash) still uses physical redo information.

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_2>.
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 the execution of a normal operation.
   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.

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.

The following actions are taken when recovering data after the system crash:

Scan log forward from last < **checkpoint** *L*> record

Repeat history by physically redoing all updates of all transactions,
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 the database to the state of crash, with committed as well as uncommitted transactions having been redone.

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

Scan log backwards, performing undo on log records of transactions found in *undo-list*.

- Transactions are rolled back as described earlier.
- 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 effect of incomplete transactions (those with neither **commit** nor **abort** log records). Recovery is now complete.

Check pointing is done as follows:

- Output all log records in memory to stable storage,
- Output to disk all modified buffer blocks,
- Output to log on stable storage a < **checkpoint** *L*> record,
- Transactions are not allowed to perform any action while checkpointing is in progress.

Another checkpoint mechanism called **Fuzzy Check pointing** allows transactions to progress once the checkpoint record is written back to a stable storage, but modified buffer blocks are not yet completely written to the stable storage. **Fuzzy check pointing** 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,*
   (a) Blocks should not be updated while being output,
   (b) Follow Write ahead log scheme: 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 **lastcheckpoint** on disk.

When recovering the database using a fuzzy checkpoint, start scan from the **checkpoint** record pointed to by **last_checkpoint.**

... [16]

- 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.

2.12 REMOTE BACKUP SYSTEMS

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

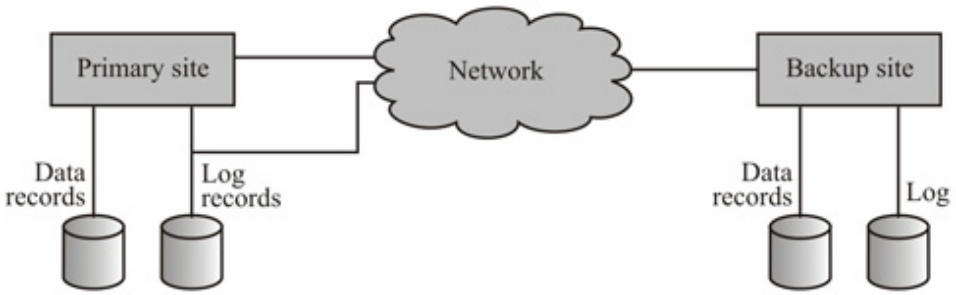


Figure 10: Remote backup system

Detection of Failure: Backup site must be able to detect when the primary site has failed. To distinguish primary site failure from link failure, we would need to maintain several communication links between the primary and remote backup.

Transfer of Control: To take over control, backup site first performs recovery using its copy of the database and all the log records it has received from the primary site. Thus, completed transactions are redone and incomplete transactions are rolled back. When the backup site takes over processing, it becomes the new primary site in order to transfer control back to the old primary site. When it recovers the database, the old primary site receive redo logs from the old backup and apply all updates locally.

Time to recover: To reduce delay in takeover, backup site periodically proceses the redo log records (in effect, performing recovery from previous database state), performs a checkpoint, and can then delete earlier parts of the log.

Hot-Spare configuration permits very fast takeover, backup continually processes redo log record as they arrive, applying the updates locally. When failure of the primary site is detected, the backup rolls back incomplete transactions, and is ready to process new transactions.

Alternative to remote backup are distributed databases with replicated data. Remote backup is faster and cheaper, but has a lower tolerance level for detecting failure.

The hot-spare configuration ensures durability of updates by delaying transaction committed until update is logged at backup. You can avoid this delay by permitting lower degrees of durability.

One-safe: A transaction commits in this scheme as soon as transaction's commit log record is written at the primary site. The problem in such a case is that, updates may not arrive at backup site before it takes over.

Two-very-safe: A transaction commits when transaction's commit log record is written at both sites primary and backup. Although this reduces the availability of the data base since transactions cannot commit if either site fails.

Two-safe: This commit protocol proceeds if both the primary and backup site are active. However, if only the primary is active, the transaction commits as soon as its commit log record is written at the primary site. This protocol provides better availability than two-very-safe, as it avoids problems of lost transactions in one-safe.

2.13 E-TRANSACTIONS

With the Internet boom more and more applications are supporting Internet based transactions. Such transactions may require a sequence of or a combination of the following activities:

- Dissemination of information using product catalogues such as information about trains in a Railway reservation system,
- Price negotiation-sometimes possible in airline reservations, but not in Railway reservation system,
- Actual order for example, booking of a train reservation,
- Payment for the order for example, payment for the booking,
- Electronic delivery if product is electronically deliverable, or else information about product tracking, for example, electronic printing of ticket.
- Communication for after sale services. For example, any other information about trains.

Such e-transactions are slightly lengthier and require a secure transaction processing method. They should be very reliable as there are many possible types of failures. More detailed discussion on these topics is available in the further readings section.

☞ Check Your Progress 3

1) Define log based recovery process.

……

……

2) How are remote back up recovery process helpful in transaction processing methods?

……

……

3) Elaborate the dual paging problem.

……

……

4) Write algorithms for various protocols discussed in this unit.

……

……

……

2.14 SUMMARY

In this unit we have discussed transaction management and various recovery management methods. We have detailed the features of transactions and log based as well as timestamp based protocols for transaction management. Two phase locking and its variants are very useful mechanisms for implementing concurrent transactions without having any concurrency related problem. But locking often leads to deadlocks. These deadlocks can be avoided if we use timestamp-based protocols. Locking can be done at multiple granularity and the index can also be locked to avoid inconsistent analysis problems. Multi-version scheme although requiring very little rollback has a high overhead. SQL supports weaker level of consistency, which requires programmer support for avoidance of concurrency related problems.

The recovery mechanism is needed in database system to take care of failures. One can use either the log based or page based recovery schemes. Buffer management is an important issue for DBMS as it affects the process of recovery. Remote backup systems allow a copy of complete data.

Formatted: Centered

2.15 SOLUTION/ANSWERS

Check Your Progress 1

1) The transaction processing methods can be:

   Batch – combining jobs in batches
   Online – The effects are known immediately
   Advanced methods may include real time transaction, weak consistency levels etc.

2) Lock based protocols are one of the major techniques to implement concurrent transaction execution. These protocols try to ensure serialisability. The pitfalls of such protocols can be:

   - Overheads due to locking
   - Locking may lead to deadlocks
   - We may either restrict concurrency – strict 2 PL or allow uncommitted data to be seen by other transactions.

3) A timestamp is a label that is allotted to a transaction when it starts. This label is normally generated as per some time unit so that transactions may be identified as per time order. They allow concurrency control but without letting transaction wait for a lock. Thus, problems such as deadlocks may be avoided.

4) The main problems of Insert and Delete are that it may result in inconsistent analysis when concurrent transactions are going on due to phantom records. One such solution to this problem may be locking the index while performing such an analysis.

5) Multiple granularity defines object at various levels like database, file, record, fields, etc.

6) Test them yourself with some data.

Check Your Progress 2

1) Multi-version technique allows the creation of multi-version of each data items as it gets updated. It also records the timestamp of updating transaction for that data. This protocol tries to make available correct data as per transaction timestamp.

2) Deadlock is a situation when two or more transactions are waiting for each other to release locks held by others. A deadlock can be detected when a cycle is detected in the wait for graph. The basic procedure of deadlock avoidance may be to roll back certain transaction as per some rule such as wound-wait or allow the transaction a maximum time to finish after which it may be rolled back.

3) Weak level of consistency in SQL allows transaction to be executed at a lower level. In such cases the programmer need to ensure that there is no concurrency related problem. SQL provides the following weak consistency levels. Read uncommitted, Read committed, and Repeatable Read.

4) It can only be implemented with more secondary storage, which are placed at different locations to take care of natural disasters or catastrophic events.

Check Your Progress 3

1) All recovery process requires redundancy. Log based recovery process records a consistent state of database and all the transaction logs after that on a stable storage. In case of any failure the stable log and the database states are used to create the consistent database state.

2) They increase the availability of the system. Also provides a situation through which we may avoid natural disaster or catastrophic failures.

3) When a page buffer under an operating system control is written back to disk swap space by operating system and is required by DBMS, this require two block transfer, hence the name dual paging problem.

4) The algorithms are given in the unit just test them.

Page 1: [1] Formatted	**RAJU**	**2/23/2007 9:50 AM**

Centered

Page 1: [2] Formatted	**RAJU**	**2/23/2007 9:50 AM**

Centered

Page 1: [3] Formatted	**RAJU**	**2/23/2007 9:50 AM**

Centered

Page 1: [4] Formatted	**RAJU**	**2/23/2007 9:50 AM**

Centered

Page 1: [5] Formatted	**RAJU**	**2/23/2007 9:50 AM**

Centered

Page 1: [6] Formatted	**RAJU**	**2/23/2007 9:50 AM**

Centered

Page 1: [7] Formatted	**RAJU**	**2/23/2007 9:50 AM**

Centered

Page 1: [8] Formatted	**RAJU**	**2/23/2007 9:50 AM**

Centered

Page 1: [9] Formatted	**RAJU**	**2/23/2007 9:50 AM**

Centered

Page 1: [10] Formatted	**RAJU**	**2/23/2007 9:50 AM**

Centered

Page 1: [11] Formatted	**RAJU**	**2/23/2007 9:50 AM**

Centered

Page 48: [12] Formatted	**RAJU**	**2/23/2007 9:50 AM**

French France

Page 48: [13] Formatted	**RAJU**	**2/23/2007 9:50 AM**

English U.S.

Page 1: [14] Formatted	**RAJU**	**2/23/2007 9:50 AM**

Centered

Page 1: [15] Formatted	**RAJU**	**2/23/2007 9:50 AM**

Centered

Page 1: [16] Formatted	**RAJU**	**2/23/2007 9:50 AM**

Centered

Page 1: [17] Formatted	**RAJU**	**2/23/2007 9:50 AM**

Centered

Page 1: [18] Formatted	**RAJU**	**2/23/2007 9:50 AM**

Centered