Concurrency Control in DBMS

Dr. Hrudaya Kumar Tripathy,
School of Computer Engineering, KIIT

Concurrency Control

- Concurrency control concept is a procedure which helps us for the management of two simultaneous processes to execute without conflicts between each other, these conflicts occur in multi user systems.
- Concurrency can simply be said to be executing multiple transactions at a time. It is required to increase time efficiency. If many transactions try to access the same data, then inconsistency arises. Concurrency control required to maintain consistency of data.
- For example, if we take ATM machines and do not use concurrency, multiple persons cannot draw money at a time in different places. This is where we need concurrency.

Advantages

- ✓ Waiting time will be decreased.
- ✓ Response time will decrease.
- ✓ Resource utilization will increase.
- ✓ System performance & Efficiency is increased.

The simultaneous execution of transactions over shared databases can create several data integrity and consistency problems.

For example, if too many people are logging in the ATM machines, serial updates and synchronization in the bank servers should happen whenever the transaction is done, if not it gives wrong information and wrong data in the database.

Main problems in using Concurrency:

- **Updates will be lost** – One transaction does some changes and another transaction deletes that change. One transaction nullifies the updates of another transaction.

- **Uncommitted Dependency or dirty read problem** – On variable has updated in one transaction, at the same time another transaction has started and deleted the value of the variable there the variable is not getting updated or committed that has been done on the first transaction this gives us false values or the previous values of the variables this is a major problem.

- **Inconsistent retrievals** – One transaction is updating multiple different variables, another transaction is in a process to update those variables, and the problem occurs is inconsistency of the same variable in different instances.

Reasons for using Concurrency control method is DBMS:

- To apply Isolation through mutual exclusion between conflicting transactions
- To resolve read-write and write-write conflict issues
- To preserve database consistency through constantly preserving execution obstructions
- The system needs to control the interaction among the concurrent transactions. This control is achieved using concurrent-control schemes.
- Concurrency control helps to ensure serializability

Example

Assume that two people who go to electronic kiosks at the same time to buy a movie ticket for the same movie and the same show time.

However, there is only one seat left in for the movie show in that particular theatre. Without concurrency control in DBMS, it is possible that both moviegoers will end up purchasing a ticket.

However, concurrency control method does not allow this to happen. Both moviegoers can still access information written in the movie seating database.

But concurrency control only provides a ticket to the buyer who has completed the transaction process first.

Concurrency control techniques

Locking:

Lock guaranties exclusive use of data items to a current transaction. It first accesses the data items by acquiring a lock, after completion of the transaction it releases the lock.

Types of Locks

The types of locks are as follows –

- ✓ **Shared Lock [Transaction can read only the data item values]**
- ✓ **Exclusive Lock [Used for both read and write data item values]**

Time Stamping:

Time stamp is a unique identifier created by DBMS that indicates relative starting time of a transaction. Whatever transaction we are doing it stores the starting time of the transaction and denotes a specific time.

This can be generated using a system clock or logical counter. This can be started whenever a transaction is started. Here, the logical counter is incremented after a new timestamp has been assigned.

Lock-Based Protocols

Lock-Based Protocols

- Locking is a procedure used to control concurrent access to data.
- Locks enable a multi-user database system to maintain the integrity of transactions by isolating a transaction from others executing concurrently.
- In this type of protocol, any transaction cannot read or write data until it acquires an appropriate lock on it.

Data items can be locked in two modes:

Shared lock or Read lock: If a transaction Ti has obtained a shared mode lock(S) on data item Q, then Ti can only read the data item Q, but cannot write on Q.

Exclusive lock or Write lock: If a transaction Ti has obtained an exclusive mode lock(X) on data item Q, then Ti can both read and write Q.

A transaction must obtain a lock on a data item before it can perform a read or write operation

Basic Rules for Locking:

- If a transaction has a read lock on a data item, it can only read the item; but cannot update its value.
- If a transaction has a read lock on a data item, other transactions can obtain read locks on the same data item, but they cannot obtain any update lock on it.
- If a transaction has a write lock on a data item, then it can both read and update the value of that data item.
- If a transaction has a write lock on a data item, then other transactions cannot obtain either a read lock or a write lock on that data item.
- A transaction requests a shared lock on data item Q by executing the Lock-S(Q) instruction.
- Similarly, a transaction can request an exclusive lock through the Lock-X(Q) instruction.
- A transaction can unlock a data item Q by the Unlock(Q) instruction.

Working of Locking

- All transactions that need to access a data item must first acquire a read lock or write lock on the data item depending on whether it is a read only operation or not.
- If the data item for which the lock is requested is not already locked, then the transaction is granted with the requested lock immediately.
- If the item is currently locked, the database system determines what kind of lock is the current one. Also, it finds out which type of lock is requested:

- *If a read lock is requested on a data item that is already under a read lock, then the request will be granted.*
- *If a write lock is requested on a data item that is already under a read lock, then the request will be denied.*
- *Similarly; if a read lock or a write lock is requested on a data item that is already under a write lock, then the request is denied and the transaction must wait until the lock is released.*

- A transaction continues to hold the lock until it explicitly releases it either during the execution or when it terminates.
- The effects of a write operation will be visible to other transactions only after the lock is released.

Schedule3

T_1	T_2
Read(A);	
Write(A);	
	Read(A);
	Write(A);
Read(B);	
Write(B);	
	Read(B);
	Write(B);

T_1	T_2	Concurrency-Control Manager
Lock-X(A)		
		Grant-X(A, T_1)
Read(A);		
Write(A);		
Unlock(A)		
	Lock-X(A)	
		Grant-X(A, T_2)
	Read(A);	
	Write(A);	
	Unlock(A)	
Lock-X(B)		
		Grant-X(B, T_1)
Read(B);		
Write(B);		
Unlock(B)		
	Lock-X(B)	
		Grant-X(B, T_2)
	Read(B);	
	Write(B);	
	Unlock(B)	

Schedule5

T_1	T_2
Read(A);	
Write(A);	
	Read(A);
Read(B);	
	Write(A);
	Read(B);
	Write(B);
Write(B);	

T_1	T_2	Concurrency-Control Manager
Lock-X(A)		
		Grant-X(A, T_1)
Read(A);		
Write(A);		
Unlock(A)		
	Lock-X(A)	
		Grant-X(A, T_2)
	Read(A);	
Lock-X(B)		
		Grant-X(B, T_1)
Read(B);		
	Write(A);	
	Unlock(A)	
	Lock-X(B)	

T_1	T_2
Read(A);	Read(A);
A:=A-100;	Read(B);
Write(A);	Display(A+B);
Read(B);	
B:=B+100;	
Write(B);	

T_1	T_2
Lock-X(A);	Lock-S(A);
Read(A);	Read(A);
A:=A-100;	Unlock(A);
Write(A);	Lock-S(B);
Unlock(A);	Read(B);
Lock-X(B);	Unlock(B);
Read(B);	Display(A+B);
B:=B+100;	
Write(B);	
Unlock(B);	

T_1	T_2	Concurrency-Control Manager
Lock-X(A)		
		Grant-X(A, T_1)
Read(A);		
A:=A-100;		
Write(A);		
Unlock(A)		
	Lock-S(A)	
		Grant-S(A, T_2)
	Read(A);	
	Unlock(A)	
	Lock-S(B)	
		Grant-S(B, T_2)
	Read(B);	
	Unlock(B)	
	Display(A+B);	
Lock-X(B)		
		Grant-X(B, T_1)
Read(B);		
B:=B+100;		
Write(B);		
Unlock(B)		

Though the concurrency control manager will not face any problem in granting the locks, the above schedule gives incorrect result for transaction T2

To solve the previous discussed problem, different alternative solutions are possible. One solution can be by delaying the unlocking process. That means the unlocking is delayed to the end of the transaction

T_1	T_2
Lock-X(A);	Lock-S(A);
Read(A);	Read(A);
A:=A-100;	Lock-S(B);
Write(A);	Read(B);
Lock-X(B);	Display(A+B);
Read(B);	Unlock(A);
B:=B+100;	Unlock(B);
Write(B);	
Unlock(A);	
Unlock(B);	

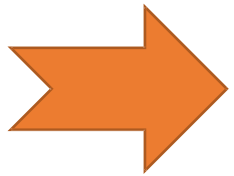
Unfortunately, this type of locking can lead to an undesirable situation

T_1	T_2	Concurrency-Control Manager
Lock-X(A)		
		Grant-X(A, T_1)
Read(A);		
A:=A-100;		
Write(A);		
	Lock-S(A)	

Since T1 is holding an exclusive-lock on A and T2 is requesting a shared-lock on A, the concurrency control manager will not grant the lock permission to T2. Thus, T2 is waiting for T1 to unlock A

T_3
Lock-X(B);
Read(B);
B:=B-100;
Write(B);
Lock-X(A);
Read(A);
A:=A+100;
Write(A);
Unlock(B);
Unlock(A);

T_2
Lock-S(A);
Read(A);
Lock-S(B);
Read(B);
Display(A+B);
Unlock(A);
Unlock(B);



T_3	T_2	Concurrency-Control Manager
Lock-X(B)		
		Grant-X(B, T_3)
Read(B);		
B:=B-100;		
Write(B);		
	Lock-S(A)	
		Grant-S(A, T_2)
	Read(A);	
	Lock-S(B);	
Lock-X(A)		

- T2 is waiting for T3 to unlock B. Similarly, T3 is waiting for T2 to unlock A. Thus, this is a situation where neither of these transactions can ever proceed with its normal execution. This type of situation is called **deadlock.**

- If we do not use locking, or if we unlock data items as soon as possible after reading or writing them, we may get **inconsistent states.**

- On the other hand, if we do not unlock a data item before requesting a lock on another data item, **deadlocks may occur.**

When a transaction requests a lock on a data item in a particular mode, and no other transaction has put a lock on the same data item in a conflicting mode, then the lock can be granted by the concurrency control manager. However, we must take some precautionary measures to avoid the following scenarios:

- Suppose a transaction T1 has a shared-mode lock on a data item, and another transaction T2 requests an exclusive-mode lock on that same data item. In this situation, T2 has to wait for T1 to release the shared-mode lock.

- Suppose, another transaction T3 requests a shared-mode lock on the same data item while T1 is holding a shared lock on it. As the lock request is compatible with lock granted to T1, so T3 may be granted the shared-mode lock. But, T2 has to wait for the release of the lock from that data item.

- At this point, T1 may release the lock, but still T2 has to wait for T3 to finish. There may be a new transaction T4 that requests a shared-mode lock on the same data item, and is granted the lock before T3 releases it.

- In such a situation, T2 never gets the exclusive-mode lock on the data item. Thus, T2 cannot progress at all and is said to be starved. This problem is called as the **starvation problem.**

We can avoid starvation of transactions by granting locks in the following manner; when a transaction Ti requests a lock on a data item Q in a particular mode M, the concurrency-control manager grants the lock provided that:

- There is no other transaction holding a lock on Q in a mode that conflicts with M
- There is no other transaction that is waiting for a lock on Q and that made its lock request before Ti

Four types of lock protocols

- Simplistic lock protocol
- Pre-claiming Lock Protocol
- Two-phase locking (2PL)
- Strict Two-phase locking (Strict-2PL)

Simple lock protocol

- It is the simplest way of locking the data while transaction.
- Simple lock protocols allow all the transactions to get the lock on the data before insert or delete or update on it.
- It will unlock the data item after completing the transaction.

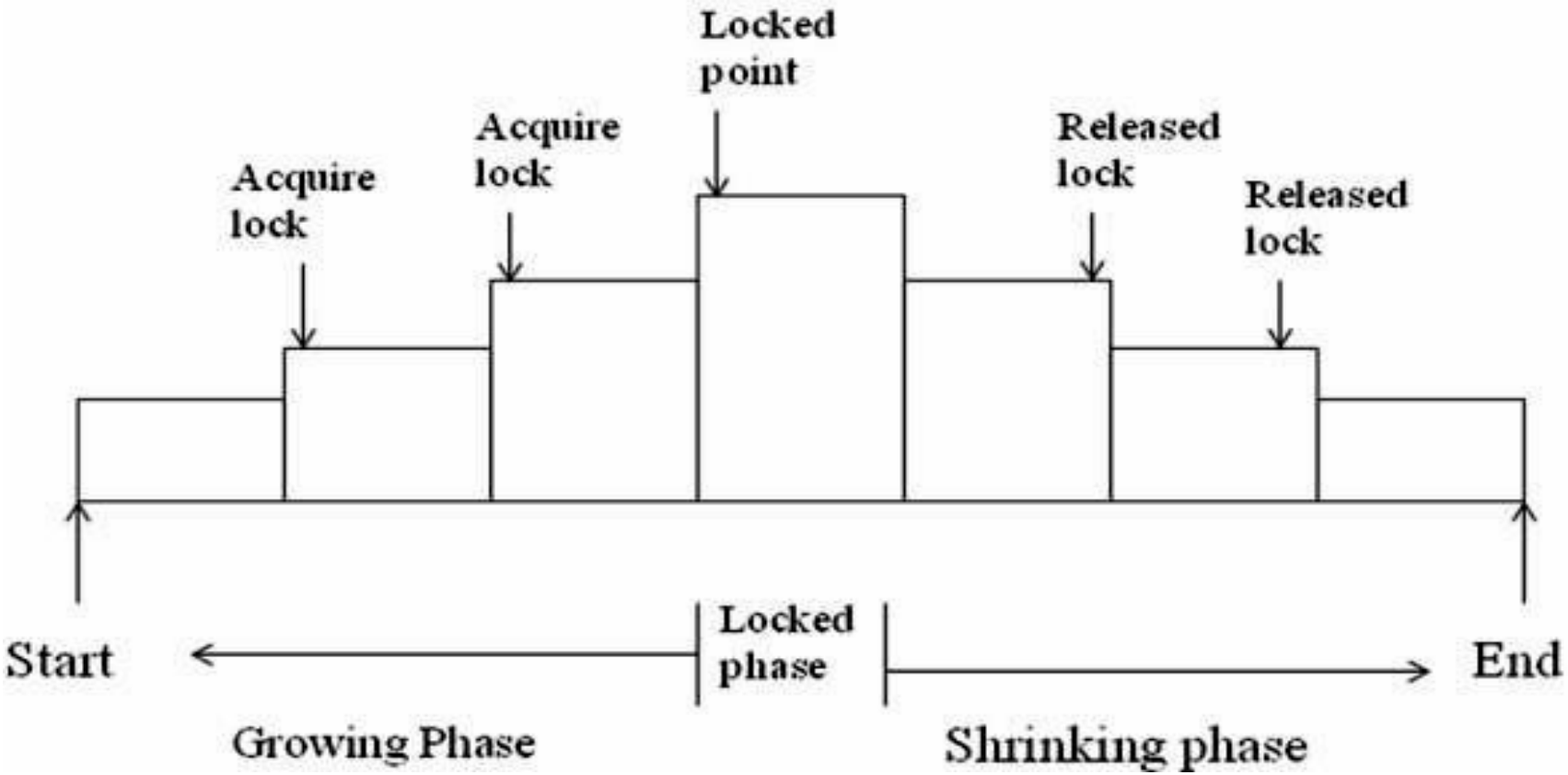
Two-Phase Locking Protocol

- **Two-phase locking protocol is a protocol which ensures serializability. This protocol requires that each transaction issues lock and unlock requests in two phases. The two phases are:**

  Growing phase: **Here, a transaction acquires all required locks without unlocking any data, i.e. the transaction may not release any lock.**

  Shrinking phase: **Here, a transaction releases all locks and cannot obtain any new lock. The point in the schedule where the transaction has obtained its final lock is called the lock point of the transaction.**

The point in the schedule where the transaction has obtained its final lock is called the lock point of the transaction. Transactions can be ordered according to their lock points



- Two transactions cannot have conflicting locks
- No unlock operation can precede a lock operation in the same transaction
- No data are affected until all locks are obtained, i.e. until the transaction is in its locked point

- Two-phase locking does not ensure freedom from deadlock.
- Along with the serializability property, the schedules should be cascadeless in nature.
- Cascading rollback is possible under two-phase locking protocol.

T_4	T_5	T_6
Lock-X(A); Read(A); Lock-S(B); Read(B); Write(A); Unlock(A);		
	Lock-X(A); Read(A); Write(A); Unlock(A);	
		Lock-S(A); Read(A);

- Cascading rollbacks can be avoided by a modification of the original two-phase locking protocol called **strict two-phase locking protocol**. In this protocol, a transaction must hold all its exclusive locks till it commits or aborts.

- **Rigorous two-phase locking protocol** is even stricter, which requires that all the locks be held until the transaction commits or aborts. In this protocol, transactions can be serialized in the order in which they commit.

- Most database systems implement either strict or rigorous two-phase locking.

Lock Conversions

- **Lock Upgrade:** This is the process in which a shared lock is upgraded to an exclusive lock
- **Lock Downgrade:** This is the process in which an exclusive lock is downgraded to a shared lock

Lock upgrading can take place only in the growing phase, where as lock downgrading can take place only in the shrinking phase

Thus, the two-phase locking protocol with lock conversions:

- ***First Phase***:
  - Can acquire a lock-S on item
  - Can acquire a lock-X on item
  - Can convert a lock-S to a lock-X (upgrade)
- ***Second Phase***:
  - Can release a lock-S
  - Can release a lock-X
  - Can convert a lock-X to a lock-S (downgrade)

T_7	T_8
Lock-S(a1);	
Read(a1);	
	Lock-S(a1);
	Read(a1);
Lock-S(a2);	
Read(a2);	
Lock-S(a3);	
Read(a3);	
	Lock-S(a3);
	Read(a3);
	Display(a1+a3);
	Unlock(a1);
	Unlock(a3);
...	
Lock-S(an);	
Read(an);	
Upgrade(a1);	
Write(a1);	

Timestamp Based Protocols

- The timestamp method for concurrency control doesn't need any locks and therefore this method is free from deadloack situation.
- Locking methods generally prevent conflicts by making transaction to wait; where as timestamp methods do not make the transactions to wait. Rather, transactions involved in a conflicting situation are simply rolled back and restarted.
- A timestamp is a unique identifier created by the database system that indicates the relative starting time of a transaction.
- Timestamps are generated either using the system clocks or by incrementing a logical counter every time a new transaction starts.
- Timestamp protocol is a concurrency control protocol in which the fundamental goal is to order the transactions globally in such a way that older transactions get priority in the event of a conflict.

- The Timestamp Ordering Protocol is used to order the transactions based on their Timestamps. The order of transaction is nothing but the ascending order of the transaction creation.
- The priority of the older transaction is higher that's why it executes first. To determine the timestamp of the transaction, this protocol uses system time or logical counter.
- The lock-based protocol is used to manage the order between conflicting pairs among transactions at the execution time. But Timestamp based protocols start working as soon as a transaction is created.
- Let's assume there are two transactions T1 and T2. Suppose the transaction T1 has entered the system at 007 times and transaction T2 has entered the system at 009 times. T1 has the higher priority, so it executes first as it is entered the system first.
- The timestamp ordering protocol also maintains the timestamp of last 'read' and 'write' operation on a data.

Basic Timestamp ordering protocol works as follows:

1. Check the following condition whenever a transaction Ti issues a **Read (X)** operation:

   - If W_TS(X) >TS(Ti) then the operation is rejected.
   - If W_TS(X) <= TS(Ti) then the operation is executed.
   - Timestamps of all the data items are updated.

2. Check the following condition whenever a transaction Ti issues a **Write(X)** operation:

   - If TS(Ti) < R_TS(X) then the operation is rejected.
   - If TS(Ti) < W_TS(X) then the operation is rejected and Ti is rolled back otherwise the operation is executed.

Where,

TS(TI) denotes the timestamp of the transaction Ti.

R_TS(X) denotes the Read time-stamp of data-item X.

W_TS(X) denotes the Write time-stamp of data-item X.

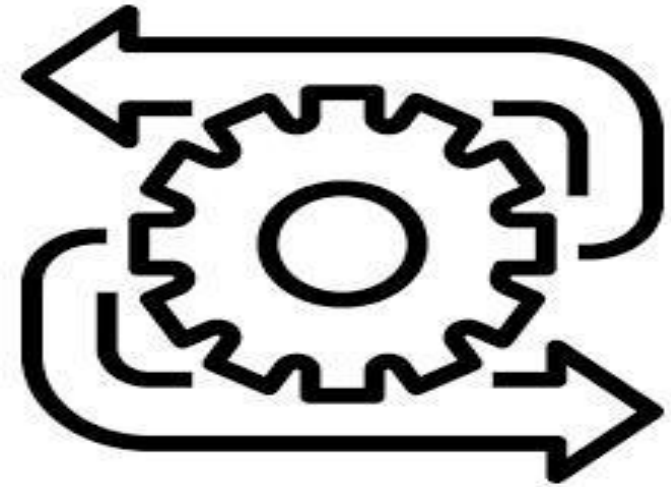
Advantages and Disadvantages of TO protocol:

- TO protocol ensures serializability since the precedence graph is as follows:

Image: Precedence Graph for TS ordering

- TS protocol ensures freedom from deadlock that means no transaction ever waits.
- But the schedule may not be recoverable and may not even be cascade- free.

Recovery System in DBMS



Transaction Failure

- The recovery manager of a database system is responsible for ensuring two important properties of transactions:
  - *Atomicity*
  - *Durability*
- It ensures the atomicity by undoing the actions of transactions that do not commit.
- It ensures the durability by making sure that all actions of committed transactions survive in case of system crashes and media failures

We generalize failure into various categories, as follows –

- **Transaction failure:**
  - *Logical error:* The transaction can not complete due to some internal error conditions, such as wrong input, data not found, overflow or resource limit exceeded.
  - System error: The transaction can not complete because of the undesirable state. However, the transaction can be re-executed at a later time.
- **System crash:** Power failure or other hardware or software failure causes the system to crash. This causes the loss of content of volatile storage and brings transaction processing to a halt. But, the content of nonvolatile storage remains intact and is not corrupted.
- **Disk failure:** A disk block loses its content as a result of either a head crash or failure during a data transfer operation. Copies of the data on other disks are used to recover from the failure.

Database Recovery

- Database recovery is the process of restoring a database to the correct state in the event of a failure.

- This service is provided by the database system to ensure that the database is reliable and remains in consistent state in case of a failure.

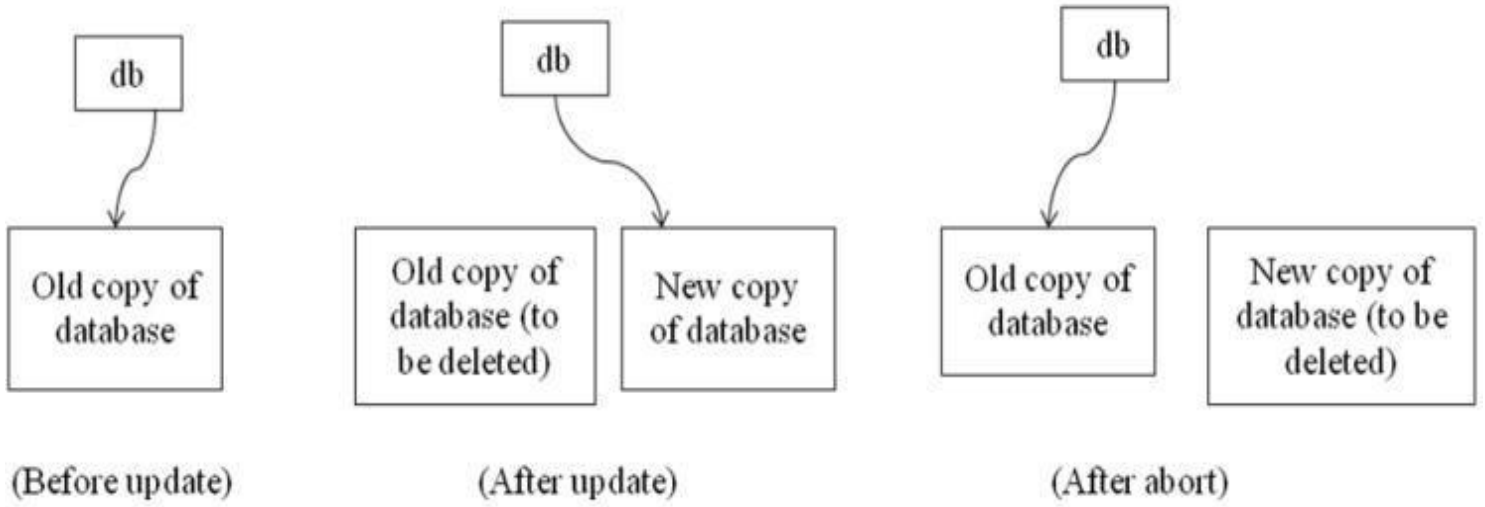
- The recovery algorithms, which ensure database consistency and transaction atomicity, consist of two parts:

  - ***Actions taken during normal transaction processing to ensure that enough information exists to allow recovery from failures.***

  - ***Actions taken after a failure to recover the database contents to a state that ensures database consistency, transaction atomicity and durability.***

Shadow-Copy Scheme:

- This scheme is based on making copies of the database called shadow copies and it assumes that only one transaction is active at a time.
- This scheme assumes that the database is simply a file on disk. A pointer called db-pointer is maintained on disk; it points to the current copy of the database.



Unfortunately, this implementation is extremely inefficient in the context of large database, since executing a single transaction requires copying the entire database.

Also, the implementation does not allow transactions to execute concurrently with one another.

Thus, this cannot be used for efficient recovery

Recovery Facilities

A database system should provide the following facilities to assist with the recovery:

- ✓ **Backup Mechanism**: It makes periodic backup copies of the database.
- ✓ **Logging Facility:** It keeps track of the current state of transactions and the database modifications.
- ✓ **Checkpoint Facility:** It enables updates to the database that are in progress to be made permanent.
- ✓ **Recovery Management:** It allows the system to restore the database to a consistent state following a failure.

Log-Based Recovery:

Log is a sequence of log records, recording all the update activities in the database. It is kept on stable storage. There are several types of log records. An update log record describes a single database write. It has the following fields:

- ***Transaction Identifier:*** This is the unique identifier of the transaction that performed the write operation
- ***Data-item Identifier:*** This is the unique identifier of the data item written. Typically, it is the location on disk of the data item
- ***Old Value:*** This is the value of the data item prior to the write operation
- ***New Value:*** This is the value that the data item will have after the write operation

Various types of log records are:

- $<T_i$ **start**$>$: Transaction T_i has started
- $<T_i, X_j, V_1, V_2>$: Transaction T_i has performed a write operation on the data item x_j. This data item x_j had value v_1 before the write, and will have value v_2 after the write
- $<T_i$ **commit**$>$: Transaction T_i has committed
- $<T_i$ **abort**$>$: Transaction T_i has aborted

When a transaction performs a write operation, it is essential that the log record for that write be created before the database is modified.

For log records to be useful for recovery from system and disk failures, the log must reside in stable storage.

The log contains a complete record of all database activities.

There are two approaches to modify the database:

- **Deferred database modification**
- **Immediate database modification**

Deferred Database Modification

- The deferred modification technique occurs if the transaction does not modify the database until it has committed.
- In this method, all the logs are created and stored in the stable storage, and the database is updated when a transaction commits.

Immediate database modification

- The Immediate modification technique occurs if database modification occurs while the transaction is still active.
- In this technique, the database is modified immediately after every operation. It follows an actual database modification.

Deferred Database Modification

The execution of transaction T_i proceeds as follows:

- Before T_i starts its execution, a record $<T_i$ start$>$ is written to the log
- A write(X) operation by T_i results in the writing of a new record to the log as $<T_i$, X, V$>$, where V is the new value of X. For this scheme, the old value is not required
- When T_i partially commits, a record $<T_i$ commit$>$ is written to the log

Case-1: Crash occurs just before $<T_1$ commit>

Database Log
$<T_1$ start>
$<T_1$,A,900>
$<T_1$,B,2100>

When the system recovers, no redo actions need to be taken, because no commit record appears in the log. The log records of the incomplete transaction T_1 can be deleted from the log

Case-2 Crash occurs just after $<T_1$ commit>

Database Log
$<T_1$ start>
$<T_1$,A,900>
$<T_1$,B,2100>
$<T_1$ commit>

When the system recovers, the operation redo(T_1) is performed since the record $<T_1$ commit> appears in the log

Case-3: Crash occurs just before $<T_2$ commit>

Database Log
$<T_1$ start>
$<T_1$,A,900>
$<T_1$,B,2100>
$<T_1$ commit>
$<T_2$ start>
$<T_2$,C, 2900>

When the system recovers, the operation redo(T_1) is performed since the record $<T_1$ commit> appears in the log. The log records of the incomplete transaction T_2 can be deleted from the log

Case-4: Crash occurs just after $<T_2$ commit>

Database Log
$<T_1$ start>
$<T_1$,A,900>
$<T_1$,B,2100>
$<T_1$ commit>
$<T_2$ start>
$<T_2$,C, 2900>
$<T_2$ commit>

When the system recovers, two commit records are found in the log. Thus, the system must perform operations redo(T_1) and redo(T_2) in the order in which their commit records appear in the log

Immediate Database Modification

The execution of transaction T_i proceeds as follows:

- Before T_i starts its execution, the system writes the record $<T_i$ start$>$ to the log
- During its execution, any write(X) operation by T_i is preceded by the writing of the appropriate new update record to the log
- When T_i partially commits, the system writes the record $<T_i$ commit$>$ to the log

Case-1: Crash occurs just before <T_1 commit>

Database Log
<T_1 start>
<T_1,A,1000,900>
<T_1,B,2000,2100>

When the system recovers, it finds the record <T_1 start> in the log, but no corresponding <T_1 commit> record. Thus, T_1 must be undone, so an undo(T_1) is performed

Case-2 Crash occurs just after <T_1 commit>

Database Log
<T_1 start>
<T_1,A,1000,900>
<T_1,B,2000,2100>
<T_1 commit>

When the system recovers, the operation redo(T_1) must be performed since the log contains both the record <T_1 start> and the record <T_1 commit>

Case-3: Crash occurs just before $<T_2$ commit>

Database Log
$<T_1$ start>
$<T_1$,A,1000,900>
$<T_1$,B,2000,2100>
$<T_1$ commit>
$<T_2$ start>
$<T_2$,C, 3000, 2900>

When the system recovers, two recovery actions need to be taken. The undo(T_2) must be performed, because the record $<T_2$ start> appears in the log, but there is no record $<T_2$ commit>. The operation redo(T_1) must be performed since the log contains both the record $<T_1$ start> and the record $<T_1$ commit>

Case-4: Crash occurs just after $<T_2$ commit>

Database Log
$<T_1$ start>
$<T_1$,A,1000,900>
$<T_1$,B,2000,2100>
$<T_1$ commit>
$<T_2$ start>
$<T_2$,C, 3000, 2900>
$<T_2$ commit>

When the system recovers, both T_1 and T_2 need to be redone, since the records $<T_1$ start> and $<T_1$ commit> appear in the log, as do the records $<T_2$ start> and $<T_2$ commit>

Checkpoints

When a system failure occurs, we must consult the log to determine those transactions that need to be redone and those that need to be undone. For this, we need to search the entire log to determine this information. There are two major difficulties with this approach:

- The search process is time consuming
- Most of the transactions that need to be redone have already written their updates into the database. Although redoing them will cause no harm, it will nevertheless cause recovery to take longer time

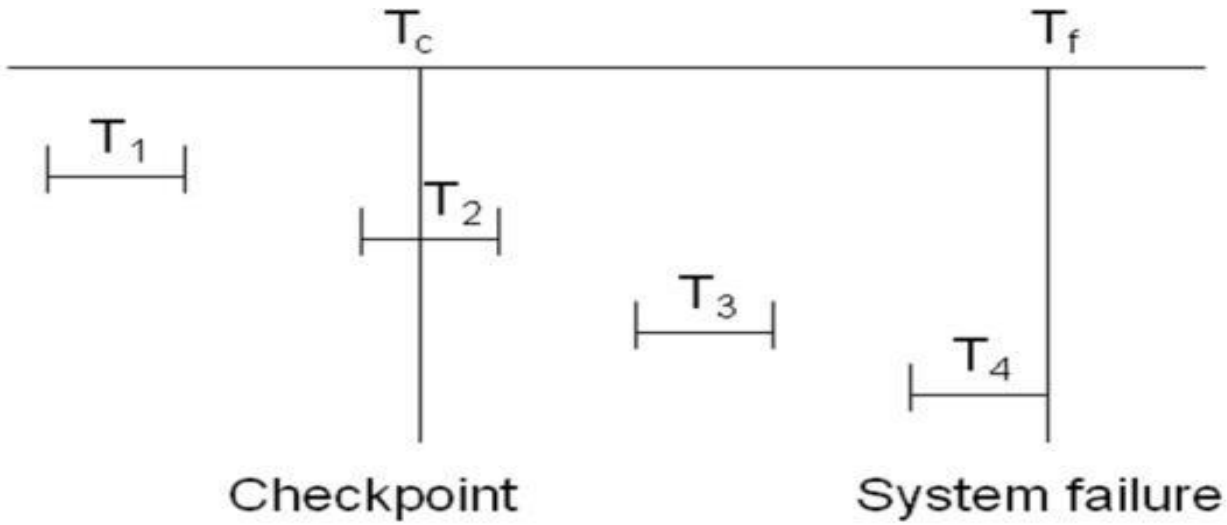
To reduce these types of overhead, checkpoints can be used

- Output onto stable storage all log records currently residing in main memory
- Output to the disk all modified buffer blocks
- Output onto stable storage a log record <checkpoint>

While a checkpoint is in progress, transactions are not allowed to perform any update actions such as writing to a buffer block or writing a log record

After a failure has occurred, the recovery scheme examines the log to determine the most recent transaction T_i that started executing before the most recent checkpoint took place

- Scan backwards from end of log to find the most recent <checkpoint> record
- Continue scanning backward till a record $<T_i$ start> is found
- Once the system has identified transaction T_i, the redo and undo operations need to be applied to only transaction T_i and all transactions T_j that started executing after transaction T_i. Let these transactions are denoted by the set T. The earlier part of the log can be ignored and can be erased whenever desired
- For all transactions starting from T_i or later with no $<T_i$ commit> record in the log, execute undo(T_i)
- Scanning forward in the log, for all transactions starting from T_i or later with a $<T_i$ commit>, execute redo(T_i)



- Transaction T_1 has to be ignored
- Transactions T_2 and T_3 have to be redone
- T_4 has to be undone

By taking checkpoints periodically, the DBMS can reduce the amount of work to be done during restart in the event of a subsequent crash


The End