Dayananda Sagar College of Engineering
Kumara Swamy Layout,Bangalore-560078
Department of Artificial Intelligence & Machine Learning

Unit -5

Transaction Processing, Concurrency Control, and Recovery: Introduction to Transaction Processing, Transaction and System Concepts, Desirable Properties of Transactions, Two-Phase Locking Techniques for Concurrency Control, Recovery Concepts, NO-UNDO/REDO Recovery Techniques based on Deferred Update, Recovery Techniques Based on Immediate Update, Shadow Paging, The ARIES Recovery Algorithm.
Mongo DB: CRUD & nesting, Indexing, Aggregation, Map reduce, Replica set, Sharding, Geospatial and GridFS

Text Book:

1. Database systems Models, Languages, Design and Application Programming, RamezElmasri and Shamkant B. Navathe, 7th Edition, 2017, Pearson.

2. Database management systems, Ramakrishnan, and Gehrke, 3rd Edition, 2014, McGraw Hill

Prepared by,
Dr.Aruna M G
Associate Professor
Department of AI&ML
DSCE
Bangalore

Transaction in DBMS

- A transaction is a program including a collection of database operations, executed as a logical unit of data processing.
- The operations performed in a transaction include one or more of database operations like insert, delete, update or retrieve data.
- It is an atomic process that is either performed into completion entirely or is not performed at all. A transaction involving only data retrieval without any data update is called read- only transaction.
- Each high level operation can be divided into a number of low level tasks or operations. For example, a data update operation can be divided into three tasks –

  - **read_item() − reads data item from storage to main memory.**
  - **modify_item() − change value of item in the main memory.**
  - **write_item() − write the modified value from main memory to storage.**

  Database access is restricted to read_item() and write_item() operations. Likewise, for all transactions, read and write forms the basic database operations.

- **Executing a read_item(X) command includes the following steps:**

1. Find the address of the disk block that contains item X.
2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer). The size of the buffer is the same as the disk block size.
3. Copy item X from the buffer to the program variable named X.

- **Executing a write_item(X) command includes the following steps:**

1. Find the address of the disk block that contains item X.
2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
3. Copy item X from the program variable named X into its correct location in the buffer.
4. Store the updated disk block from the buffer back to disk (either immediately or at some later point in time).

- **Types of Users**

Single User	Multi User
A DBMS is single-user if at most one user at a time can use the system	A DBMS is multi-user if many users can use the system and hence access the database concurrently.
Single user databases do not have multiprogramming thus, single CPU can only execute at most one process at a time.	Multiple users can access databases and use computer systems simultaneously because of the concept of Multiprogramming.

Transactions and Database Items

- A transaction is an executing program that forms a logical unit of database processing.
- A transaction includes one or more database access operations—these can include insertion, deletion, modification (update), or retrieval operations.
- The database operations that form a transaction can either be embedded within an application program or they can be specified interactively via a high level query language such as SQL.
- One way of specifying the transaction boundaries is by specifying explicit begin transaction and end transaction statements in an application program; in this case, all database access operations between the two are considered as forming one transaction.

- A single application program may contain more than one transaction if it contains several transaction boundaries. If the database operations in a transaction do not update the database but only retrieve data, the transaction is called a read-only transaction; otherwise it is known as a read-write transaction.

- **Database Model :**

The database model that is used to present transaction processing concepts is simple when compared to the data models that we discussed earlier in the book,such as the relational model or the object model.

A database is basically represented as a collection of named data items. The size of a data item is called its granularity. A data item can be a database record, but it can also be a larger unit such as a whole disk block, or even a smaller unit such as an individual field (attribute) value of some record in the database.
The transaction processing concepts we discuss are independent of the data item granularity (size) and apply to data items in general. Each data item has a unique name, but this name is not typically used by the programmer; rather, it is just a means to uniquely identify each data item.

- **Operations in Transaction**

The main operations in a transaction are-

1. Read Operation
2. Write Operation

1. Read Operation

- Read operation reads the data from the database and then stores it in the buffer in main memory.
- Example : Read(A) instruction will read the value of A from the database and will store it in the buffer in main memory.

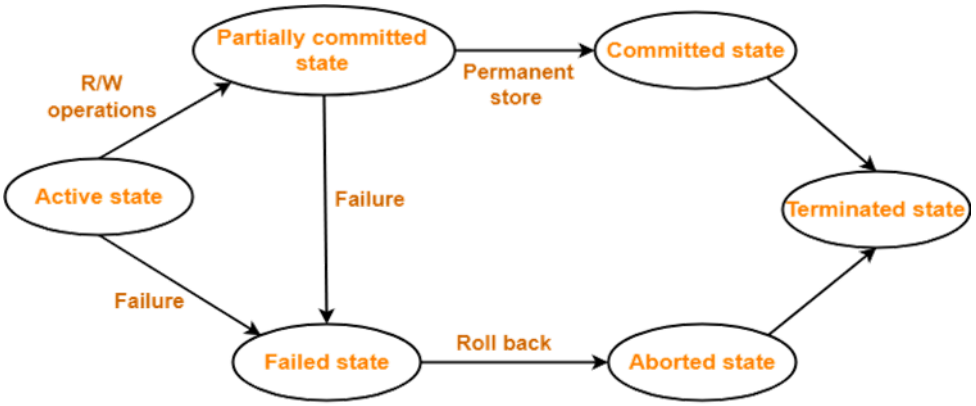
2. Write Operation

- Write operation writes the updated data value back to the database from the buffer.
- Example : Write(A) will write the updated value of A from the buffer to the database.

- **Transaction States**

A transaction goes through many different states throughout its life cycle.
These states are called as transaction states.
Transaction states are as follows-

1. Active state
2. Partially committed state
3. Committed state
4. Failed state
5. Aborted state
6. Terminated state



Transaction States in DBMS

1. Active State

- This is the first state in the life cycle of a transaction.
- A transaction is called in an active state as long as its instructions are getting executed.
- All the changes made by the transaction now are stored in the buffer in main memory.

2. Partially Committed State

- After the last instruction of transaction has executed, it enters into a partially committed state.
- After entering this state, the transaction is considered to be partially committed.
- It is not considered fully committed because all the changes made by the transaction are still stored in the buffer in main memory.

3. Committed State

- After all the changes made by the transaction have been successfully stored into the database, it enters into a committed state.
- Now, the transaction is considered to be fully committed.

NOTE

- After a transaction has entered the committed state, it is not possible to roll back the transaction.
- This is because the system is updated into a new consistent state.

- The only way to undo the changes is by carrying out another transaction called as compensating transaction that performs the reverse operations.

4. Failed State

When a transaction is getting executed in the active state or partially committed state and some failure occurs due to which it becomes impossible to continue the execution, it enters into a failed state.

5. Aborted State

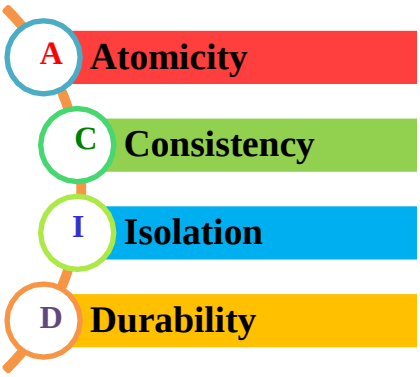
- After the transaction has failed and entered into a failed state, all the changes made by it have to be undone.
- To undo the changes made by the transaction, it becomes necessary to roll back the transaction.
- After the transaction has rolled back completely, it enters into an aborted state.

6. Terminated State

- This is the last state in the life cycle of a transaction.
- After entering the committed state or aborted state, the transaction finally enters into a terminated state where its life cycle finally comes to an end.

ACID Properties in DBMS

- It is important to ensure that the database remains consistent before and after the transaction.
- To ensure the consistency of database, certain properties are followed by all the transactions occurring in the system.
- These properties are called as ACID Properties of a transaction.



1. Atomicity

- This property ensures that either the transaction occurs completely or it does not occur at all.
- In other words, it ensures that no transaction occurs partially.
- That is why, it is also referred to as “All or nothing rule“.
- It is the responsibility of Transaction Control Manager to ensure atomicity of the transactions.

2. Consistency

- This property ensures that integrity constraints are maintained.
- In other words, it ensures that the database remains consistent before and after the transaction.
- It is the responsibility of DBMS and application programmer to ensure consistency of the database.

3. Isolation

- This property ensures that multiple transactions can occur simultaneously without causing any inconsistency.
- During execution, each transaction feels as if it is getting executed alone in the system.
- A transaction does not realize that there are other transactions as well getting executed parallel.
- A change made by a transaction becomes visible to other transactions only after they are written in the memory.
- The resultant state of the system after executing all the transactions is same as the state that would be achieved if the transactions were executed serially one after the other.
- It is the responsibility of concurrency control manager to ensure isolation for all the transactions.

4. Durability

- This property ensures that all the changes made by a transaction after their successful executions are written successfully to the disk.
- It also ensures that these changes exist permanently and are never lost even if there occurs a failure of any kind.
- It is the responsibility of recovery manager to ensure durability in the database.

Problems with Concurrent Execution

- In a database transaction, the **two main operations are READ and WRITE operations**. So, there is a need to manage these two operations in the concurrent execution of the transactions as if these operations are not performed in an interleaved manner, and the data may become inconsistent.
- So, the following problems occur with the Concurrent Execution of the operations:

1. Concurrency Problems in DBMS

- When multiple transactions execute concurrently in an uncontrolled or unrestricted manner, then it might lead to several problems.
- Such problems are called as concurrency problems.
- The five concurrency problems that can occur in the database are:

1. **Temporary Update Problem**
2. **Incorrect Summary Problem**
3. **Lost Update Problem**
4. **Unrepeatable Read Problem**
5. **Phantom Read Problem**

1. Dirty Read Problem or Temporary update (W-R Conflict)

It occurs when one transaction updates an item and fails. But the updated item is used by another transaction before the item is changed or reverted back to its last value.

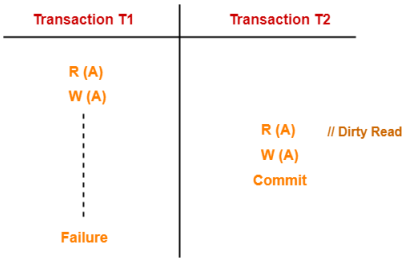
Reading the data written by an uncommitted transaction is called as dirty read. This read is called as dirty read because

- There is always a chance that the uncommitted transaction might roll back later.
- Thus, uncommitted transaction might make other transactions read a value that does not even exist.
- This leads to inconsistency of the database.

NOTE :

- Dirty read does not lead to inconsistency always.
- It becomes problematic only when the uncommitted transaction fails and roll backs later due to some reason.

Example 1 :



1. T1 reads the value of A.
2. T1 updates the value of A in the buffer.
3. T2 reads the value of A from the buffer.
4. T2 writes the updated the value of A.
5. T2 commits.
6. T1 fails in later stages and rolls back.

In this example,

- ✓ T2 reads the dirty value of A written by the uncommitted transaction T1.
- ✓ T1 fails in later stages and roll backs.
- ✓ Thus, the value that T2 read now stands to be incorrect.
- ✓ Therefore, database becomes inconsistent.

Example 2: In the below example, if transaction 1 fails for some reason then X will revert back to its previous value. But transaction 2 has already read the incorrect value of X.

T1	T2
read_item(X) X = X - N write_item(X)	
	read_item(X) X = X + M write_item(X)
read_item(Y)	

Example 3: Consider two transactions TX and TY in the below diagram performing read/write operations on account A where the available balance in account A is $300:

Time	T_x	T_y
t_1	READ (A)	—
t_2	A = A + 50	—
t_3	WRITE (A)	—
t_4	—	READ (A)
t_5	SERVER DOWN ROLLBACK	—

DIRTY READ PROBLEM

- At time t1, transaction T_X reads the value of account A, i.e., $300.
- At time t2, transaction T_X adds $50 to account A that becomes $350.
- At time t3, transaction T_X writes the updated value in account A, i.e., $350.
- Then at time t4, transaction T_Y reads account A that will be read as $350.
- Then at time t5, transaction T_X rollbacks due to server problem, and the value changes back to $300 (as initially).
- But the value for account A remains $350 for transaction T_Y as committed, which is the dirty read and therefore known as the Dirty Read Problem.

2. Unrepeatable Read Problem (R-R Conflict)

Also known as Inconsistent Retrievals Problem that occurs when in a transaction, two different values are read for the same database item.

This problem occurs when a transaction gets to read unrepeated i.e. different values of the same variable in its different read operations even when it has not updated its value.

Transaction T1	Transaction T2
R (X)	
	R (X)
W (X)	
	R (X) // Unrepeated Read

1. T1 reads the value of X (= 10 say).
2. T2 reads the value of X (= 10).
3. T1 updates the value of X (from 10 to 15 say) in the buffer.
4. T2 again reads the value of X (but = 15).

In this example,

- ✓ T2 gets to read a different value of X in its second reading.
- ✓ T2 wonders how the value of X got changed because according to it, it is running in isolation.

Example 2: Consider two transactions, TX and TY, performing the read/write operations on account A, having an available balance = $300. The diagram is shown below:

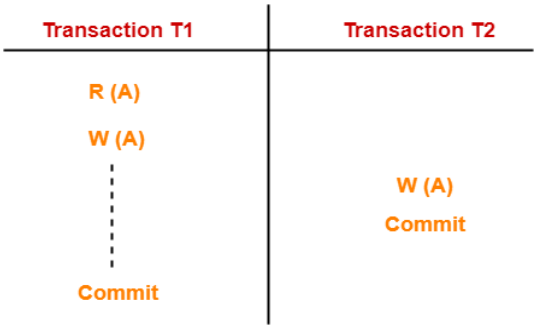
Time	T_x	T_y
t_1	READ (A)	—
t_2	—	READ (A)
t_3	—	A = A + 100
t_4	—	WRITE (A)
t_5	READ (A)	—

UNREPEATABLE READ PROBLEM

- At time t1, transaction T_X reads the value from account A, i.e., $300.
- At time t2, transaction T_Y reads the value from account A, i.e., $300.
- At time t3, transaction T_Y updates the value of account A by adding $100 to the available balance, and then it becomes $400.
- At time t4, transaction T_Y writes the updated value, i.e., $400.
- After that, at time t5, transaction T_X reads the available value of account A, and that will be read as $400.
- It means that within the same transaction T_X, it reads two different values of account A, i.e., $ 300 initially, and after updation made by transaction T_Y, it reads $400. It is an unrepeatable read and is therefore known as the Unrepeatable read problem.

3. Lost Update Problem(W-W Conflict)

This problem occurs when multiple transactions execute concurrently and updates from one or more transactions get lost.



1. T1 reads the value of A (= 10 say).
2. T2 updates the value to A (= 15 say) in the buffer.
3. T2 does blind write A = 25 (write without read) in the buffer.
4. T2 commits.
5. When T1 commits, it writes A = 25 in the database.

In this example,

- ✓ T1 writes the over written value of X in the database.
- ✓ Thus, update from T1 gets lost.

2. NOTE:

- This problem occurs whenever there is a write-write conflict.
- In write-write conflict, there are two writes one by each transaction on the same data item without any read in the middle.

Example 2: Consider two transactions, TX and TY, performing the read/write operations on account A, having an available balance = $300. The diagram is shown below:

Time	T_x	T_y
t_1	READ (A)	—
t_2	A = A - 50	
t_3	—	READ (A)
t_4	—	A = A + 100
t_5	—	—
t_6	WRITE (A)	—
t_7		WRITE (A)

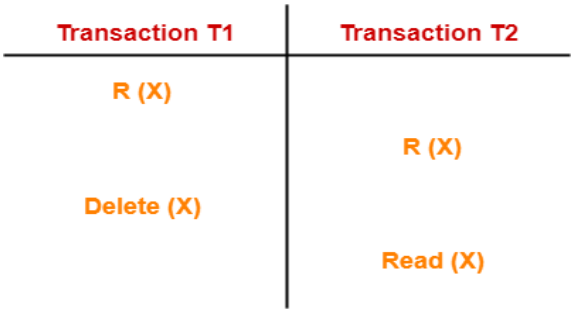
LOST UPDATE PROBLEM

- At time t1, transaction T_X reads the value of account A, i.e., $300 (only read).
- At time t2, transaction T_X deducts $50 from account A that becomes $250 (only deducted and not updated/write).
- Alternately, at time t3, transaction T_Y reads the value of account A that will be $300 only because T_X didn't update the value yet.
- At time t4, transaction T_Y adds $100 to account A that becomes $400 (only added but not updated/write).
- At time t6, transaction T_X writes the value of account A that will be updated as $250 only, as T_Y didn't update the value yet.
- Similarly, at time t7, transaction T_Y writes the values of account A, so it will write as done at time t4 that will be $400. It means the value written by T_X is lost, i.e., $250 is lost.

Hence data becomes incorrect, and database sets to inconsistent.

4. Phantom Read Problem

The phantom read problem occurs when a transaction reads a variable once from the buffer and when it tries to read that same variable again, an error occurs saying that the variable does not exist.



1. T1 reads X.
2. T2 reads X.
3. T1 deletes X.
4. T2 tries reading X but does not find it.

In this example,

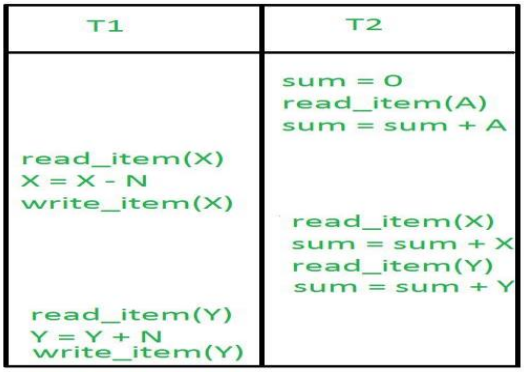
- ✓ T2 finds that there does not exist any variable X when it tries reading X again.
- ✓ T2 wonders who deleted the variable X because according to it, it is running in isolation.

In the above example, once transaction 2 reads the variable X, transaction 1 deletes the variable X without transaction 2's knowledge. Thus, when transaction 2 tries to read X, it is not able to do it.

5. Incorrect Summary Problem:

Consider a situation, where one transaction is applying the aggregate function on some records while another transaction is updating these records. The aggregate function may calculate some values before the values have been updated and others after they are updated.

Example:



In the above example, transaction 2 is calculating the sum of some records while transaction 1 is updating them. Therefore the aggregate function may calculate some values before they have been updated and others after they have been updated.

Avoiding Concurrency Problems

- To ensure consistency of the database, it is very important to prevent the occurrence of four problems discussed.
- Concurrency Control Protocols help to prevent the occurrence of above problems and maintain the consistency of the database.

Concurrency Control Protocols

The concurrency control protocols ensure the *atomicity, consistency, isolation, durability* and *serializability* of the concurrent execution of the database transactions. Therefore, these protocols are categorized as:

- Lock Based Concurrency Control Protocol
- Time Stamp Concurrency Control Protocol
- Validation Based Concurrency Control Protocol

Transaction Support in SQL

With SQL, there is no explicit Begin_Transaction statement. Transaction initiation is done implicitly when particular SQL statements are encountered. However, every transaction must have an explicit end statement, which is either a COMMIT or a ROLLBACK. Every transaction has certain characteristics attributed to it. These characteristics are specified by a SET

TRANSACTION statement in SQL. The characteristics are the *access mode*, the *diagnostic area size*, and the *isolation level*.

1. Concurrency Control Techniques

Concurrency Control is the management procedure that is required for controlling concurrent execution of the operations that take place on a database.
But before knowing about concurrency control, we should know about concurrent execution.

2. Concurrent Execution in DBMS

- In a **multi-user system**, multiple users can access and use the **same database at one time, which is known as the concurrent execution of the database.**
- It means that the same database is executed simultaneously on a multi-user system by different users.
- While working on the database transactions, there occurs the requirement of using the database by multiple users for **performing different operations**, and in that case, concurrent execution of the database is performed.
- The thing is that the simultaneous execution that is performed should be done in an **interleaved manner**, and **no operation should affect the other executing operations**, thus maintaining the consistency of the database.
- Thus, on making the concurrent execution of the transaction operations, there occur several challenging problems that need to be solved.

3. Lock-Based Protocol

In this type of protocol, any transaction cannot read or write data until it acquires an appropriate lock on it. There are two types of lock:

1. Shared lock:

- It is also known as a **Read-only lock**. In a shared lock, the data item can only read by the transaction.
- It can be shared between the transactions because when the transaction holds a lock, then it can't update the data on the data item.

2. Exclusive lock:

- In the exclusive lock, the data item can be **both reads as well as written** by the transaction.
- This lock is exclusive, and in this lock, multiple transactions do not modify the same data simultaneously.

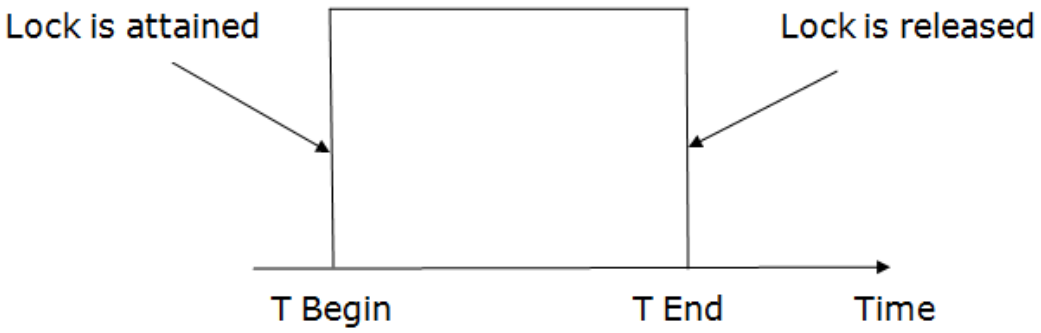
There are four types of lock protocols available:

1. Simplistic lock protocol

It is the simplest way of locking the data while transaction. Simplistic lock-based 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.

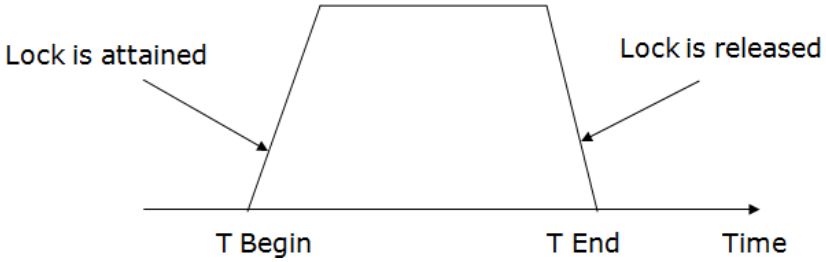
2. Pre-claiming Lock Protocol

- Pre-claiming Lock Protocols evaluate the transaction to list all the data items on which they need locks.
- Before initiating an execution of the transaction, it requests DBMS for all the lock on all those data items.
- If all the locks are granted then this protocol allows the transaction to begin. When the transaction is completed then it releases all the lock.
- If all the locks are not granted then this protocol allows the transaction to rolls back and waits until all the locks are granted.



3. Two-phase locking (2PL)/ **Two-Phase Locking Techniques for Concurrency Control**

- The two-phase locking protocol divides the execution phase of the transaction into three parts.
  - In the first part, when the execution of the transaction starts, it seeks permission for the lock it requires.
  - In the second part, the transaction acquires all the locks. The third phase is started as soon as the transaction releases its first lock.
  - In the third phase, the transaction cannot demand any new locks. It only releases the acquired locks.



There are two phases of 2PL:

Growing phase: In the growing phase, a new lock on the data item may be acquired by the transaction, but none can be released.

Shrinking phase: In the shrinking phase, existing lock held by the transaction may be released, but no new locks can be acquired.

In the below example, if lock conversion is allowed then the following phase can happen:

1. Upgrading of lock (from S(a) to X (a)) is allowed in growing phase.
2. Downgrading of lock (from X(a) to S(a)) must be done in shrinking phase.

	T1	T2
0	LOCK-S(A)	
1		LOCK-S(A)
2	LOCK-X(B)	
3	——	——
4	UNLOCK(A)	
5		LOCK-X(C)
6	UNLOCK(B)	
7		UNLOCK(A)
8		UNLOCK(C)
9	——	——

The following way shows how unlocking and locking work with 2-PL.

3. Transaction T1:

- **Growing phase:** from step 1-3
- **Shrinking phase:** from step 5-7
- **Lock point:** at 3

4. Transaction T2:

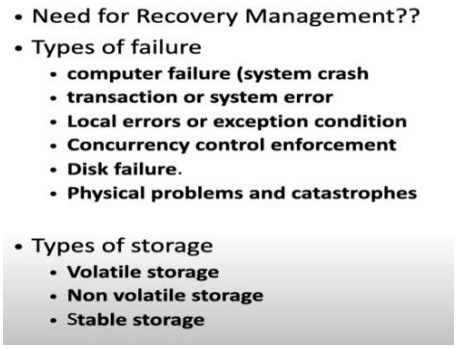
- **Growing phase:** from step 2-6
- **Shrinking phase:** from step 8-9
- **Lock point:** at 6

Database Recovery Techniques

Database systems, like any other computer system, are subject to failures but the data stored in it must be available as and when required. When a database fails it must possess the facilities for fast recovery. It must also have atomicity i.e. either transactions are completed successfully and committed (the effect is recorded permanently in the database) or the transaction should have no effect on the database.

There are both automatic and non-automatic ways for both, backing up of data and recovery from any failure situations. The techniques used to recover the lost data due to system crash, transaction errors, viruses; catastrophic failure, incorrect commands execution etc. are database recovery techniques. So to prevent data loss recovery techniques based on deferred update and immediate update or backing up data can be used.

Recovery techniques are heavily dependent upon the existence of a special file known as a **system log**. It contains information about the start and end of each transaction and any updates which occur in the **transaction**. The log keeps track of all transaction operations that affect the values of database items. This information is needed to recover from transaction failure.



- **The log is kept on disk start_transaction(T):** This log entry records that transaction T startsthe execution.
- **read_item(T, X)**: This log entry records that transaction T reads the value of database item X.
- **write_item(T, X, old_value, new_value):** This log entry records that transaction T changesthe value of the database item X from old_value to new_value. The old value is sometimes known as a before an image of X, and the new value is known as an afterimage of X.
- **commit(T):** This log entry records that transaction T has completed all accesses to the database successfully and its effect can be committed (recorded permanently) to the database.
- **abort(T):** This records that transaction T has been aborted.
- **checkpoint:** Checkpoint is a mechanism where all the previous logs are removed from the system and stored permanently in a storage disk. Checkpoint declares a point before whichthe DBMS was in consistent state, and all the transactions were committed.

A transaction T reaches its **commit** point when all its operations that access the database have been executed successfully i.e. the transaction has reached the point at which it willnot **abort** (terminate without completing). Once committed, the transaction is permanently recorded in the database. Commitment always involves writing a commit entry to the log and writing the log to disk.

At the time of a system crash, item is searched back in the log for all transactions T that have written a start_transaction(T) entry into the log but have not written a commit(T) entry yet; these transactions may have to be rolled back to undo their effect on the database during the recovery process.

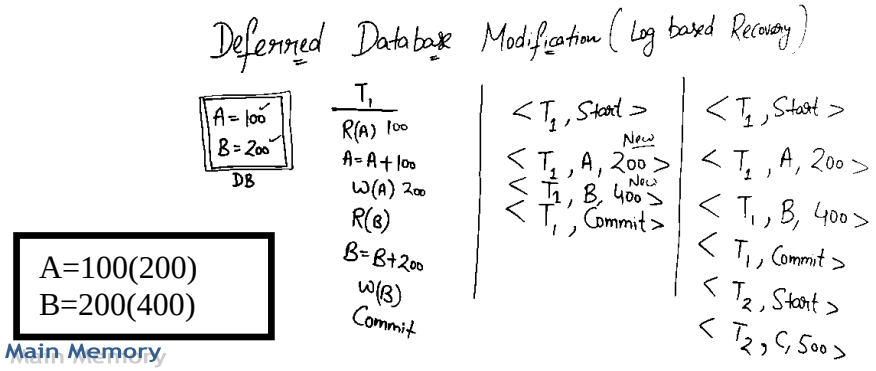
Recovery Based on Log Records

- **Transaction identifier**
- **Data-item identifier**
- **Old value**
- **New value**
- update log record **<*Ti*, *Xj*, *V1*, *V2*>,**
- <*Ti* start>. Transaction *Ti* has started.
- • <*Ti* commit>. Transaction *Ti* has committed.
- • <*Ti* abort>. Transaction *Ti* has aborted.

- **Undoing** (Backward Recovery)– If a transaction crashes, then the recovery manager may undo transactions i.e. reverse the operations of a transaction. This involves examining a transaction for the log entry**write_item(T, x, old_value, new_value)** and setting the value of item x in the database to old- value.There are two major techniques for recovery from non-catastrophic transaction failures:deferred updates and immediate updates.

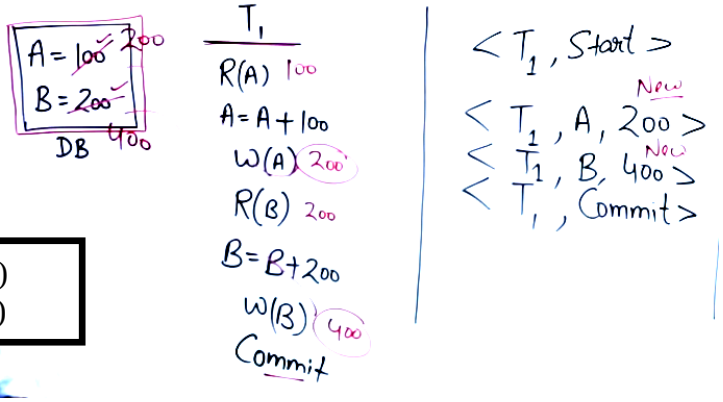
- **Deferred update –** This technique does not physically update the database on disk until a transaction has reached **its commit point**. Before reaching commit, all transaction updates are recorded in the **local transaction** workspace. If a transaction fails before reaching its commit point, it will not have changed the database in any way so **UNDO is not needed**. It may be necessary **to REDO (farward Recovery)** the effect of the operations that are recorded in the local transaction workspace, because their effect may not yet have been written in the database. Hence, a deferred update is also known as the **No-undo/redo algorithm.**

  Case1: before commit the value is updated in main memory (MM) when write operation is performed and when commit is applied then it update the value in DB/disk

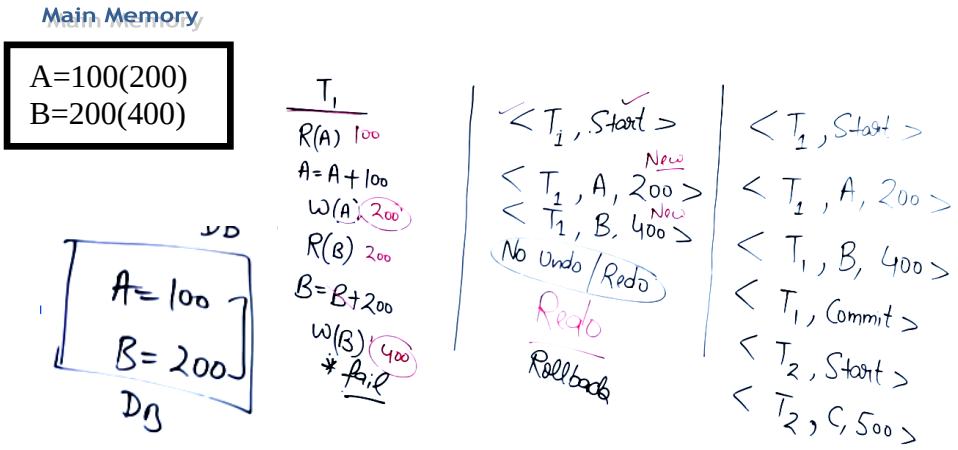


Note : recovery manager use log file/system log to check transaction details when failure occures. He does not check from DB/disk to recovery the data.

Case2: before commit the value is updated in main memory (MM) when write operation is performed and after commit is applied i.e. before writing the updated value in DB/disk a failure is occurred. The recovery manager recovers value from log file i.e 200 & 400 writes into disk.



Case3: before commit the value is updated in main memory (MM) when write operation is performed and before commit is applied a failure is occurred. The recovery manager recovers value from log file i.e 200 & 400 . No undo/ UNDO operation is not required because the value in disk is not changed.



T0: A transaction that transfers $50 from account A to account B.
T1: A transaction that withdraws $100 from account C
A:$1000 B: $ 2000 C:$700

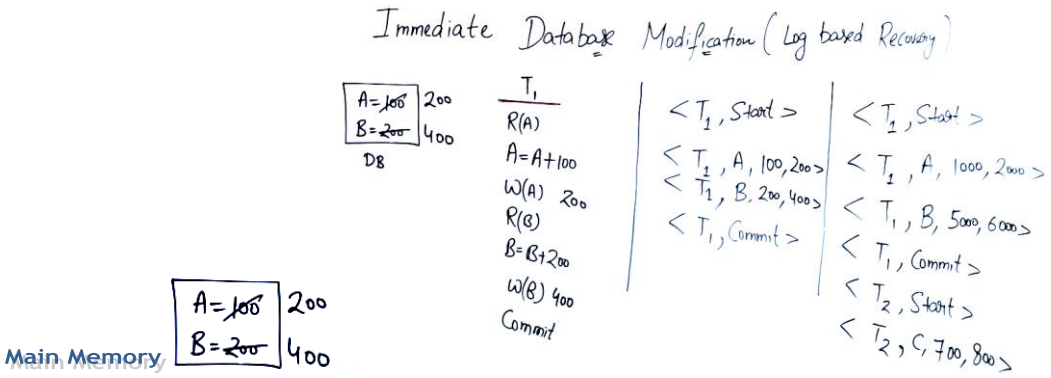
T0:		T1:	
	read(A);		read(C);
	A:=A-50;		C:=C-100;
	write(A);		write(C);
	read(B);		
	B:=B+50;		
	write(B);		

<T0 start>
<T0,A,950>
<T0,B,2050>
<T0 commit>
<T1 start>
<T1,C,600>
<T1 commit>

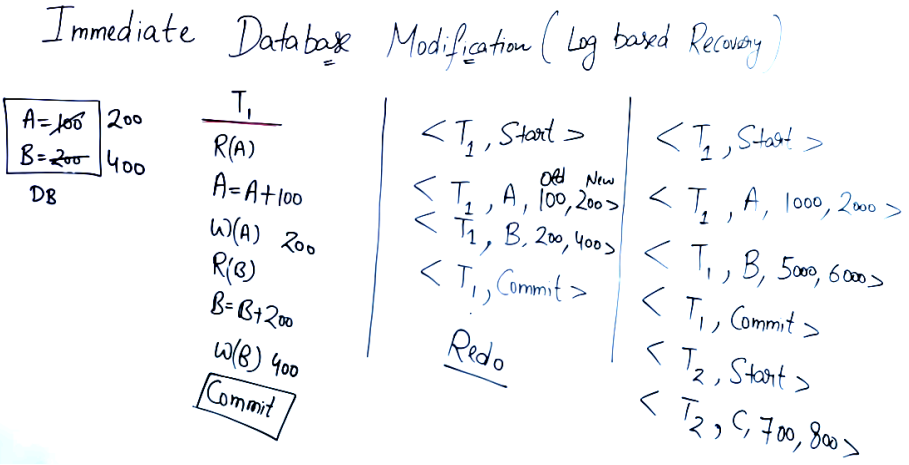
- **Immediate update** – In the immediate update, the database may be updated by some operations of a transaction before the transaction reaches its commit point. However, these operations are recorded in a log on disk before they are applied to the database, making recovery still possible. If a transaction fails to reach its commit point, the effect of its

operation must be undone i.e. the transaction must be rolled back hence we require both undo and redo. This technique is known as **undo/redo algorithm.**

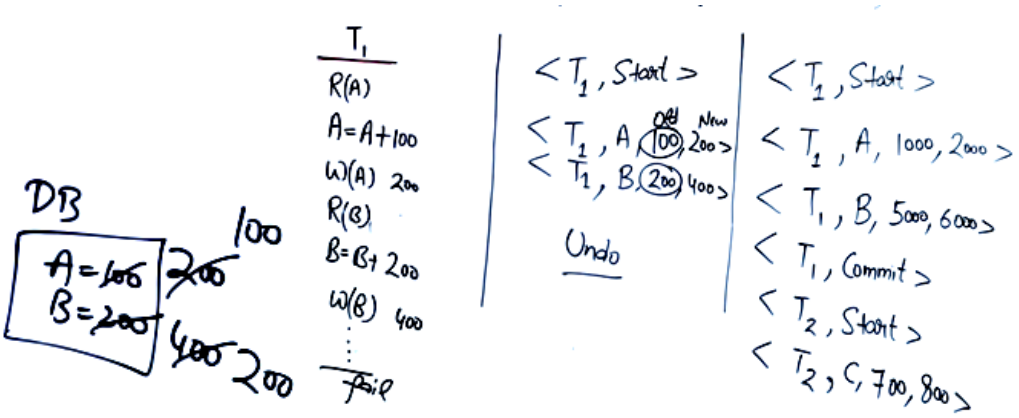
Case1: before commit the value is updated in main memory (MM) and DB/disk

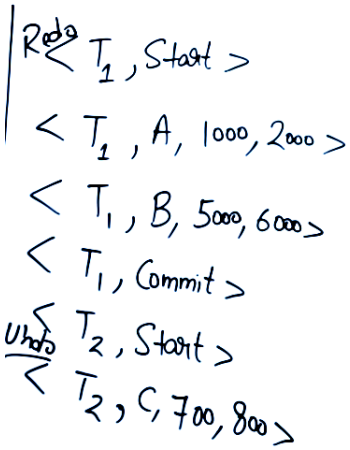


Case2: before commit the value is updated in main memory (MM) when write operation is performed and after commit is applied a failure is occurred. The recovery manager recovers value from log file i.e (old value and new value) 200 & 400. REDO operation is required because the value disk is changed.



Case3: before commit the value is updated in main memory (MM) and Disk.when write operation is performed and before commit is applied a failure is occurred. The recovery manager recovers value from log file i.e (old value and new value) 100 & 200. UNDO operation is required because the value in disk is changed with new value therefore to change into old value.



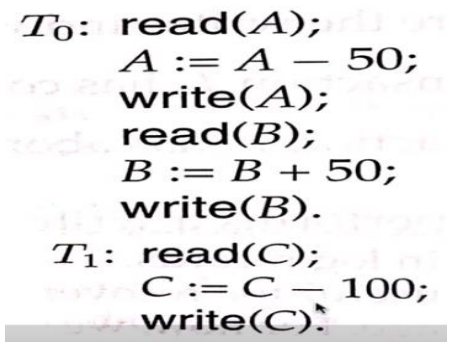


- **Caching/Buffering –** In this one or more disk pages that include data items to be updated are cached into main memory buffers and then updated in memory before being written back to disk. A collection of **in-memory buffers called the DBMS cache** is kept under control of DBMS for holding these buffers. A directory is used to keep track of which database items are in the buffer. A **dirty bit** is associated with each buffer, which **is 0 if the buffer** is not modified else **1 if modified**.

- **Shadow paging –** It provides atomicity and durability. A directory with n entries is constructed, where the ith entry points to the ith database page on the link. When a transaction began executing the current directory is copied into a shadow directory. When a page is to be modified, a shadow page is allocated in which changes are made and when it is ready to become durable, all pages that refer to original are updated to refer new replacement page.

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

- Example A= 1000 B=2000 and C= 700



Log for Deferred
DB Modification
No Undo-redo

<T_0 start>
<T_0, A, 950>
<T_0, B, 2050>
<T_0 commit>
<T_1 start>
<T_1, C, 600>
<T_1 commit>

Log for Immediate
DB Modification
Undo-redo

<T_0 start>
<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>
<T_0 commit>
<T_1 start>
<T_1, C, 700, 600>
<T_1 commit>

3.Transaction Actions That Do Not Affect the Database

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

4. NO-UNDO/REDO Recovery Based on Deferred Update

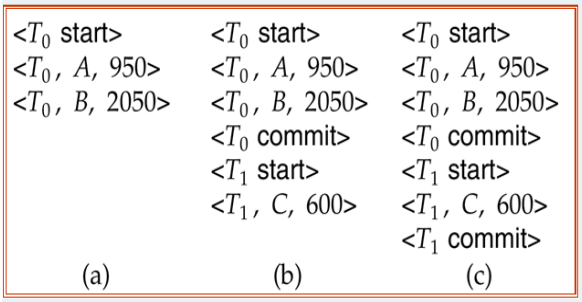
The idea behind deferred update is to defer or postpone any actual updates to the database on disk until the transaction completes its execution successfully and reaches its commit point.

During transaction execution, the updates are recorded only in the log and in the cache buffers. After the transaction reaches its commit point and the log is force-written to disk, the updates are recorded in the database. If a transaction fails before reaching its commit point, there is no need to undo any operations because the transaction has not affected the database on disk in any way. Therefore, only REDO-**type log entries** are needed in the log, which include the **newvalue** (AFIM) of the item written by a write operation. The UNDO-**type log entries** are not neededsince no undoing of operations will be required during recovery. Although this may simplify the recovery process, it cannot be used in practice unless transactions are short and each transaction changes few items. For other types of transactions, there is the potential for running out of buffer space because transaction changes must be held in the cache buffers until the commit point.

We can state a typical deferred update protocol as follows:

1. A transaction cannot change the database on disk until it reaches its commit point.
2. A transaction does not reach its commit point until all its REDO-type log entries are recorded in the log *and* the log buffer is force-written to disk.

1. Briefly discuss deferred database modification technique.



Add lock and unlock instructions to transactions T1 and T2, so that they observe the two-phase locking protocol. Can the execution of these transactions result in a deadlock?

Above is the scenario as it appears at three instances of time. Assume that crash has been occurred what the system supposed to do in all three situations.

5. Procedure RDU_M (NO-UNDO/REDO with checkpoints).

Use two lists of transactions maintained by the system: the committed transactions *T* since the last checkpoint (**commit list**), and the active transactions *T* (**active list**). REDO all the WRITE operations of the committed transactions from the log, *in the order in which they were written into the log.* The transactions that are active and did not commit are effectively canceled and must be resubmitted.

The REDO procedure is defined as follows:

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

Figure 23.2 illustrates a timeline for a possible schedule of executing transactions. When the checkpoint was taken at time t_1, transaction T_1 had committed, whereas transactions T_3 and T_4 had not. Before the system crash at time t_2, T_3 and T_2 were committed but not T_4 and T_5. According to the RDU_M method, there is no need to redo the write_item operations of transaction T_1—or any transactions committed before the last checkpoint time t_1. The write_item operations of T_2 and T_3 must be redone, however, because both transactions reached their commit points after the last checkpoint. Recall that the log is force-written before committing a transaction. Transactions T_4 and T_5 are ignored: They are effectively canceled or rolled back because none of their write_item operations were recorded in the database on disk under the deferred update protocol.

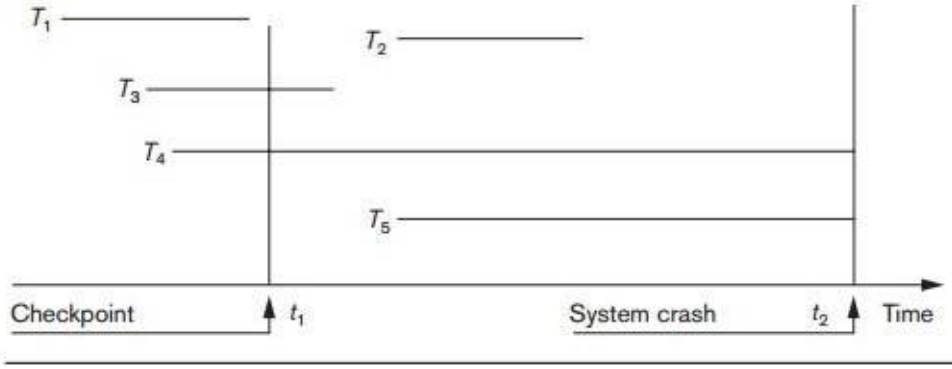


Figure 23.2
An example of a recovery timeline to illustrate the effect of checkpointing.

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

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

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

Figure 23.3 shows an example of recovery for a multiuser system that utilizes the recovery and concurrency control method just described.

6. Recovery Techniques Based on Immediate Update

In these techniques, when a transaction issues an update command, the database on disk can be updated *immediately*, without any need to wait for the transaction to reach its commit point. Notice that it is *not a requirement* that every update be

(a)

T_1	T_2	T_3	T_4
read_item(A)	read_item(B)	read_item(A)	read_item(B)
read_item(D)	write_item(B)	write_item(A)	write_item(B)
write_item(D)	read_item(D)	read_item(C)	read_item(A)
	write_item(D)	write_item(C)	write_item(A)

(b)

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

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

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

Figure 23.3
An example of recovery using deferred update with concurrent transactions. (a) The READ and WRITE operations of four transactions. (b) System log at the point of crash.

applied immediately to disk; it is just possible that some updates are applied to disk *before the transaction commits*.

Provisions must be made for *undoing* the effect of update operations that have been applied to the database by a *failed transaction.* This is accomplished by rolling back the transaction and undoing the effect of the transaction's write_item operations. Therefore, the UNDO-**type log entries**, which include the **old value** (BFIM) of the item, must be stored in the log. Because UNDO can be needed during recovery, these methods follow a **steal strategy** for deciding when updated main memory buffers can be written back to disk (see Section 23.1.3). Theoretically, we can distinguish two main categories of immediate update algorithms. If the recovery technique ensures that all updates of a transaction are recorded in the database on disk *before the transaction commits*, there is never a need to REDO any operations of committed transactions. This is called the UNDO/NO-

REDO **recovery algorithm**. In this method, all updates by a transaction must be recorded on disk *before the transaction commits*, so that REDO is never needed. Hence, this method must utilize the **force strategy** for deciding when updated main memory buffers are written back to disk (see Section 23.1.3).

If the transaction is allowed to commit before all its changes are written to the data-base, we have the most general case, known as the UNDO/REDO **recovery algorithm**. In this case, the **steal/no-force strategy** is applied (see Section 23.1.3). This is also the most complex technique. We will outline an UNDO/REDO recovery algo-rithm and leave it as an exercise for the reader to develop the UNDO/NO-REDO vari-ation. In Section 23.5, we describe a more practical approach known as the ARIES recovery technique.

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

Procedure RIU_M **(**UNDO/REDO **with checkpoints).**

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

The UNDO procedure is defined as follows:

Procedure UNDO **(**WRITE_OP**).** Undoing a write_item operation write_op consists ofexamining its log entry [write_item, *T*, *X*, old_value, new_value] and set-ting the value of item *X* in the database to old_value, which is the before image (BFIM). Undoing a numberof write_item operations from one or more trans-actions from the log must proceed in the *reverse order* from the order in which the operations were written in the log.

As we discussed for the NO-UNDO/REDO procedure, step 3 is more efficiently done by starting from the *end of the log* and redoing only *the last update of each item X*.

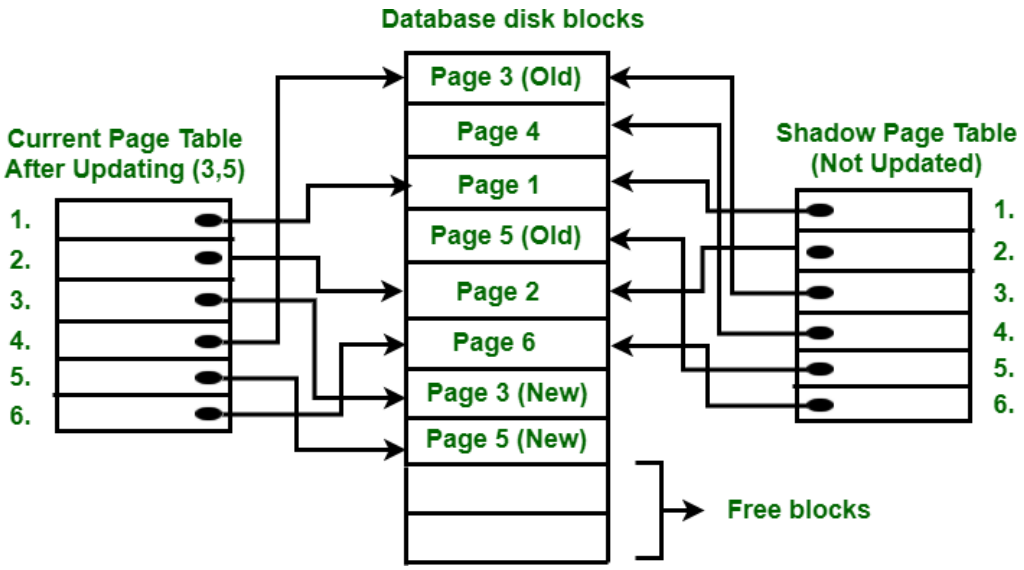
Whenever an item is redone, it is added to a list of redone items and is not redone again. A similar procedure can be devised to improve the efficiency of step 2 so that an item can be undone at most

once during recovery. In this case, the earliest UNDO is applied first by scanning the log in the forward direction (starting from the beginning of the log). Whenever an item is undone, it is added to a list of undone items and is not undone again.

7. Shadow Paging

It is a recovery technique that is used to recover database. In this recovery technique, database is considered as made up of fixed size of logical units of storage which are referred as **pages.** Pages are mapped into physical blocks of storage, with help of the **page table** which allow one entry for each logical page of database. This method uses two-page tables named **current page table** and **shadow page table**.

The entries which are present in current page table are used to point to most recent database pages on disk. Another table i.e., Shadow page table is used when the transaction starts which is copying current page table. After this, shadow page table gets saved on disk and current page table is going to be used for transaction. Entries present in current page table may be changed during execution but in shadow page table it never get changed. After transaction, both tables become identical.
This technique is also known as **Cut-of-Place updating.**



To understand concept, consider above figure. In this 2 write operations are performed on page3 and 5. Before start of write operation on page 3, current page table points to old page 3. When write operation starts following steps are performed :

1. Firstly, search start for available free block in disk blocks.
2. After finding free block, it copies page 3 to free block which is represented by Page 3 (New).
3. Now current page table points to Page 3 (New) on disk but shadow page table points to old page 3 because it is not modified.
4. The changes are now propagated to Page 3 (New) which is pointed by current page table.

8. COMMIT Operation :

To commit transaction following steps should be done :

9. All the modifications which are done by transaction which are present in buffers are transferred to physical database.
10. Output current page table to disk.
11. Disk address of current page table output to fixed location which is in stable storage containing address of shadow page table. This operation overwrites address of old shadow page table. With this current page table becomes same as shadow page table and transaction is committed.

Failure :

If system crashes during execution of transaction but before commit operation, With this, it is sufficient only to free modified database pages and discard current page table. Before execution of transaction, state of database get recovered by reinstalling shadow page table.

If the crash of system occur after last write operation then it does not affect propagation of changes that are made by transaction. These changes are preserved and there is no need to perform redo operation.

Advantages :

- This method requires fewer disk accesses to perform operation.
- In this method, recovery from crash is inexpensive and quite fast.
- There is no need of operations like- Undo and Redo.

Disadvantages :

- Due to location change on disk due to update database it is quite difficult to keep related pages in database closer on disk.
- During commit operation, changed blocks are going to be pointed by shadow page table which have to be returned to collection of free blocks otherwise they become accessible.
- The commit of single transaction requires multiple blocks which decreases execution speed.
- To allow this technique to multiple transactions concurrently it is difficult.

The ARIES Recovery Algorithm

- The ARIES Recovery Algorithm is based on:
  - **WAL** (Write Ahead Logging)
  - **Repeating history during redo**:
    - ARIES will retrace all actions of the database system prior to the crash to reconstruct the database state when the crash occurred.
  - **Logging changes during undo**:
    - It will prevent ARIES from repeating the completed undo operations if a failure occurs during recovery, which causes a restart of the recovery process.

The ARIES Recovery Algorithm (contd.)

- The ARIES recovery algorithm consists of three steps:
    1. **Analysis**: step identifies the dirty (updated) pages in the buffer and the set of transactions active at the time of crash. The appropriate point in the log where redo is to start is also determined.
    2. **Redo**: necessary redo operations are applied.
    3. **Undo**: log is scanned backwards and the operations of transactions active at the time of crash are undone in reverse order.
- **The Log and Log Sequence Number (LSN)**
    1. A log record is written for:
        - (a) data update
        - (b) transaction commit
        - (c) transaction abort
        - (d) undo
        - (e) transaction end
    2. In the case of undo a compensating log record is written.
- **The Log and Log Sequence Number (LSN)**
    1. A unique LSN is associated with every log record.
        - LSN increases monotonically and indicates the disk address of the log record it is associated with.
        - In addition, each data page stores the LSN of the latest log record corresponding to a change for that page.
    2. A log record stores
        - (a) the previous LSN of that transaction
        - (b) the transaction ID
        - (c) the type of log record.
- **The Log and Log Sequence Number (LSN)**
- A log record stores:
    1. Previous LSN of that transaction: It links the log record of each transaction. It is like a back pointer points to the previous record of the same transaction
    2. Transaction ID
    3. Type of log record
- For a write operation the following additional information is logged:
    1. Page ID for the page that includes the item
    2. Length of the updated item
    3. Its offset from the beginning of the page
    4. BFIM of the item
    5. AFIM of the item
- The **Transaction table** and the **Dirty Page table**
    1. For efficient recovery following tables are also stored in the log during checkpointing:

- **Transaction table**: Contains an entry for each active transaction, with information such as transaction ID, transaction status and the LSN of the most recent log record for the transaction.
- **Dirty Page table**: Contains an entry for each dirty page in the buffer, which includes the page ID and the LSN corresponding to the earliest update to that page.

- Checkpointing
  1. A checkpointing does the following:
     - Writes a begin_checkpoint record in the log
     - Writes an end_checkpoint record in the log. With this record the contents of transaction table and dirty page table are appended to the end of the log.
     - Writes the LSN of the begin_checkpoint record to a special file. This special file is accessed during recovery to locate the last checkpoint information.
  2. To reduce the cost of checkpointing and allow the system to continue to execute transactions, ARIES uses "fuzzy checkpointing".
- The following steps are performed for recovery
  1. **Analysis phase**: Start at the begin_checkpoint record and proceed to the end_checkpoint record. Access transaction table and dirty page table are appended to the end of the log. Note that during this phase some other log records may be written to the log and transaction table may be modified. The analysis phase compiles the set of redo and undo to be performed and ends.
  2. **Redo phase**: Starts from the point in the log up to where all dirty pages have been flushed, and move forward to the end of the log. Any change that appears in the dirty page table is redone.
  3. **Undo phase**: Starts from the end of the log and proceeds backward while performing appropriate undo. For each undo it writes a compensating record in the log.
- The recovery completes at the end of undo phase.

(a)

Lsn	Last_lsn	Tran_id	Type	Page_id	Other_information
1	0	T_1	update	C	...
2	0	T_2	update	B	...
3	1	T_1	commit		...
4	begin checkpoint				
5	end checkpoint				
6	0	T_3	update	A	...
7	2	T_2	update	C	...
8	7	T_2	commit		...

(b)

TRANSACTION TABLE

Transaction_id	Last_lsn	Status
T_1	3	commit
T_2	2	in progress

DIRTY PAGE TABLE

Page_id	Lsn
C	1
B	2

(c)

TRANSACTION TABLE

Transaction_id	Last_lsn	Status
T_1	3	commit
T_2	8	commit
T_3	6	in progress

DIRTY PAGE TABLE

Page_id	Lsn
C	1
B	2
A	6

Figure 19.6
An example of recovery in ARIES. (a) The log at point of crash. (b) The Transaction and Dirty Page Tables at time of checkpoint. (c) The Transaction and Dirty Page Tables after the analysis phase.