Database Management Systems(BCS403)

Module 4

G C DIVYA, CSE,JITD

Module – 4 Transaction Processing

Contents:

Transaction Processing

- Introduction to Transaction Processing,
- Transaction and System concepts,
- Desirable properties of Transactions,
- Characterizing schedules based on recoverability,
- Characterizing schedules based on Serializability,
- Transaction support in SQL.

Module – 4 Transaction Processing

Single-User versus Multiuser Systems

- A DBMS is **single-user** if at most one user at a time can use the system, Single-user DBMSs are mostly restricted to personal computer systems.
- A DBMS is **multiuser** if many users can use the system, and hence access the database concurrently.
- **For example,** an airline reservations system is used by hundreds of users and travel agents concurrently. Database systems used in banks, insurance agencies, stock exchanges, supermarkets, and many other applications are multiuser systems.

Module – 4 Transaction Processing

Single-User versus Multiuser Systems

- In these systems, hundreds or thousands of users are typically operating on the database by submitting transactions concurrently to the database system.

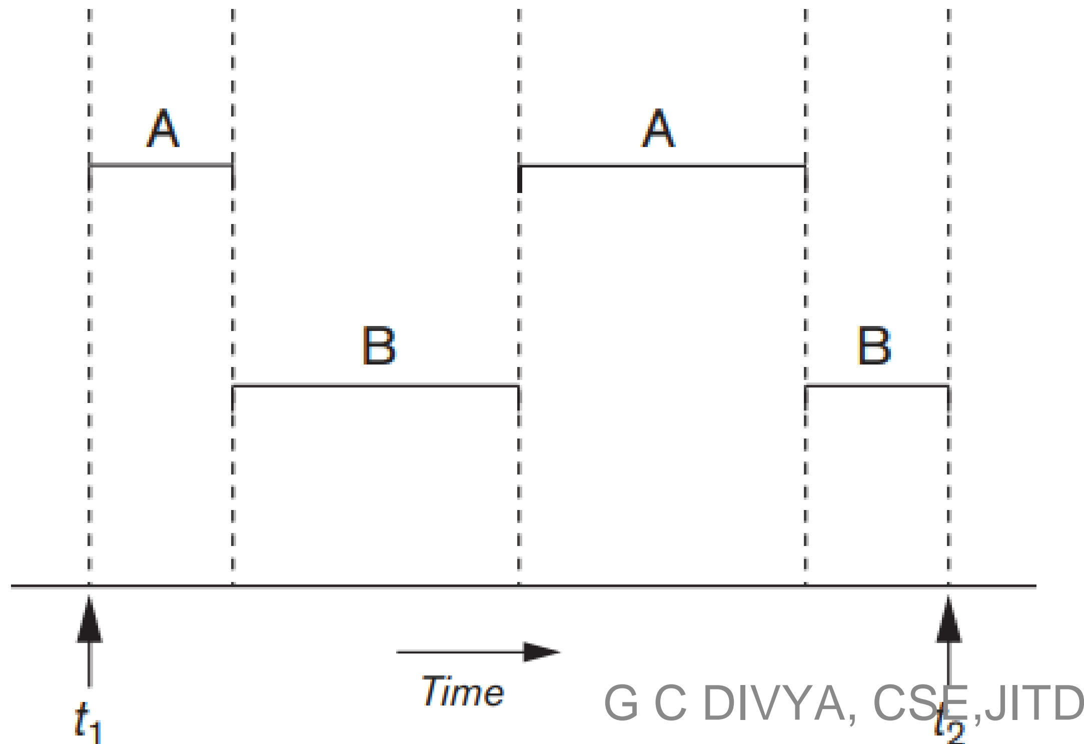


Figure: processes A and B are operating on the system in interleaved fashion.

Module – 4 Transaction Processing

Single-User versus Multiuser Systems

- A single central processing unit (CPU) can only execute at most one process at a time.
- However, **multiprogramming operating systems** execute more processes at approximately same time in interleaved fashion.
- Interleaving keeps the CPU busy when a process requires an input or output (I/O) operation, such as reading a block from disk.
- The CPU is switched to execute another process rather than remaining idle during I/O time. Interleaving also prevents a long process from delaying other processes.

Module – 4 Transaction Processing

Single-User versus Multiuser Systems

- If the computer system has multiple hardware processors (CPUs), parallel processing of multiple processes is possible.

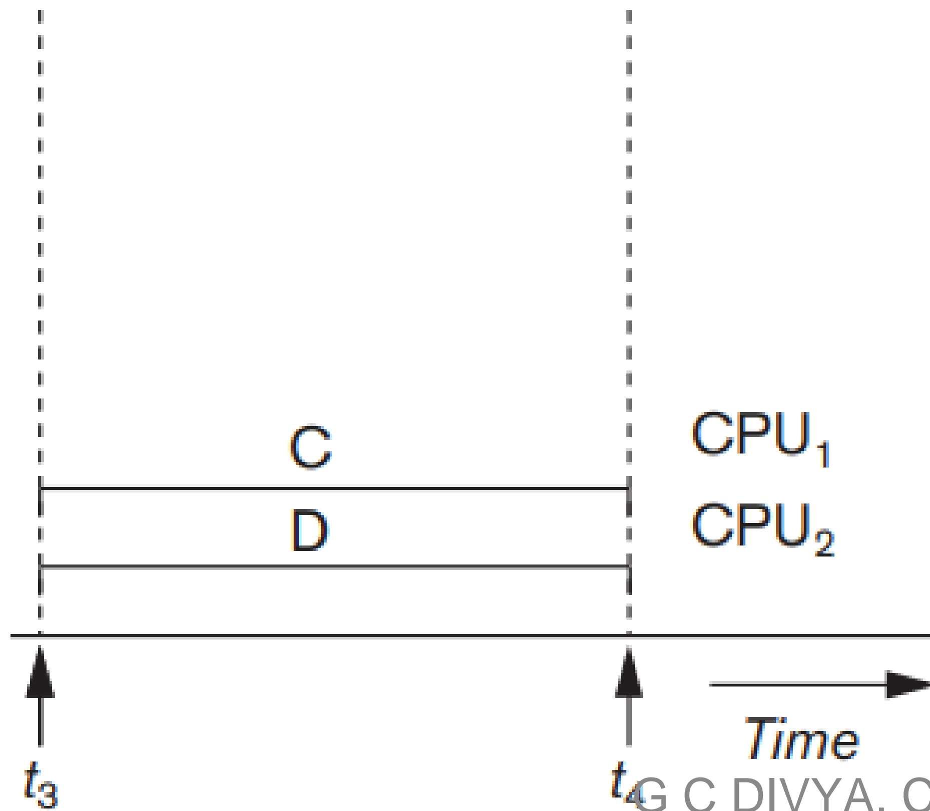


Figure: processes C and D are operating on the system in parallel.

Module – 4 Transaction Processing

Transactions, Database Items, Read and Write Operations

- **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.
- All database access operations can be specified in application programs between **begin transaction** and **end transaction** statements, called transaction boundary.
- 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**.

Module – 4 Transaction Processing

Transactions, Database Items, Read and Write Operations

The **basic database access operations** that a transaction can include are as follows:

- **read_item(X):** Reads a database item named X into a program variable.
- **write_item(X):** Writes the value of program variable X into the database item named X.

read_item(X):

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.

G C DIVYA, CSE,JITD

Module – 4 Transaction Processing

Transactions, Database Items, Read and Write Operations

3. Copy item X from the buffer to the program variable named X.

write_item(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).

Why Concurrency Control Is Needed

- Several problems can occur when concurrent transactions execute in an uncontrolled manner.
- Consider a **airline reservations database** in which a record is stored for each airline flight. Each record includes the number of reserved seats on that flight as a named (uniquely identifiable) data item, among other information.
- Transaction T1 that transfers N reservations from one flight whose number of reserved seats is stored in the database item named X to another flight whose number of reserved seats is stored in the database item named Y.
- Transaction T2 that just reserves M seats on the first flight (X) referenced in transaction T1.

Why Concurrency Control Is Needed

T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);
read_item(Y);
$Y := Y + N$;
write_item(Y);
	read_item(X);
	$X := X + M$;
	write_item(X);

- The transactions T1 and T2 are specific executions of the programs that refer to the specific flights whose numbers of seats are stored in data items X and Y in the database.

Module – 4 Transaction Processing

Why Concurrency Control Is Needed

When the transactions **T1 and T2 are executed concurrently** on this flight database, the **following problems may encounter**.

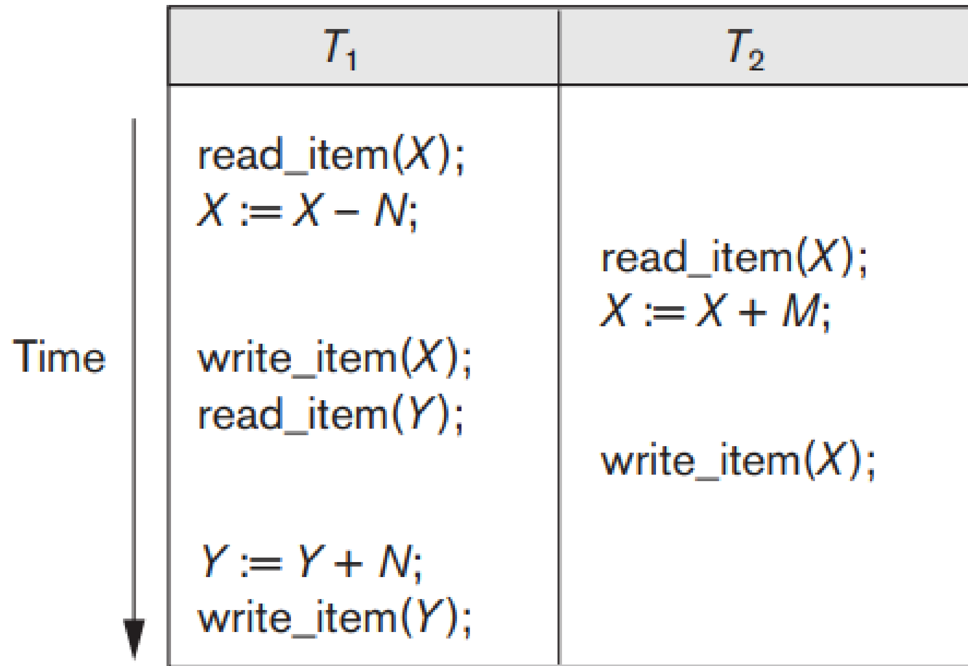
1. The Lost Update Problem.
2. The Temporary Update (or Dirty Read) Problem.
3. The Incorrect Summary Problem.
4. The Unrepeatable Read Problem.

The Lost Update Problem

- This problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database items incorrect.

Module – 4 Transaction Processing

Why Concurrency Control Is Needed



Item X has an incorrect value because its update by T1 is lost (overwritten by T2).

Suppose that transactions T1 and T2 are submitted at approximately the same time, and suppose that their operations are interleaved as shown in figure above, then the final value of item X is incorrect because T2 reads the value of X before T1 changes it in the database, and hence the updated value resulting from T1 is lost.

Why Concurrency Control Is Needed

The Temporary Update (or Dirty Read) Problem

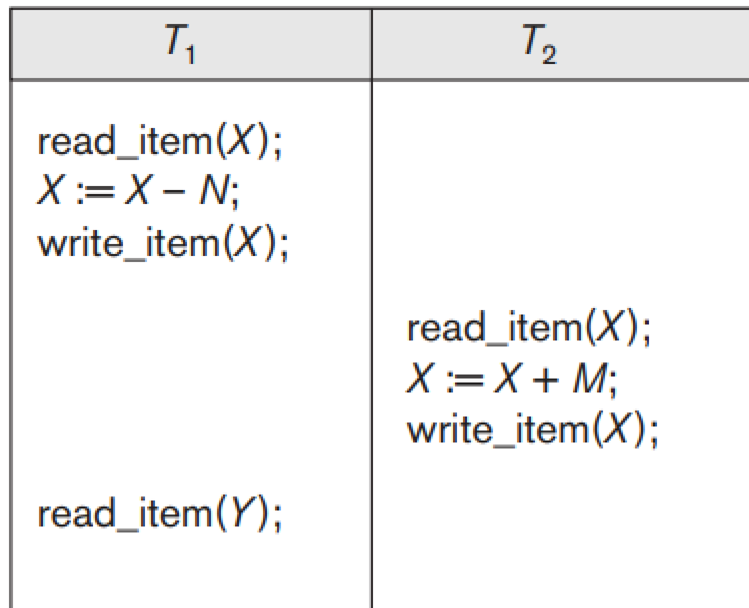
- This problem occurs when one transaction updates a database item and then the transaction fails for some reason.
- Meanwhile, the updated item is accessed (read) by another transaction before it is changed back (or rolled back) to its original value.
- In our example where T1 updates item X and then fails before completion, so the system must roll back X to its original value.
- But before roll back , the transaction T2 reads the temporary value of X, which will not be recorded permanently in the database because of the failure of T1.

Module – 4 Transaction Processing

Why Concurrency Control Is Needed

The Temporary Update (or Dirty Read) Problem

- The value of item X that is read by T2 is called **dirty data** because it has been created by a transaction that has not completed and committed yet, hence this problem is also known as the **dirty read problem.**



Transaction T1 fails and must change the value of X back to its old value; meanwhile T2 has read the temporary incorrect value of X.

Module – 4 Transaction Processing

Why Concurrency Control Is Needed

The Incorrect Summary Problem.

- If one transaction is calculating an aggregate summary function on a number of database items while other transactions are updating some of these items, the aggregate function may calculate some values before they are updated and others after they are updated.
- In our example suppose that a transaction T3 is calculating the total number of reservations on all the flights; meanwhile, transaction T1 is executing in interleaved fashion, then the result of T3 will be off by N because T3 reads the value of X after N seats have been subtracted from it but reads the value of Y before those N seats have been added to it.

Module – 4 Transaction Processing

Why Concurrency Control Is Needed

The Incorrect Summary Problem.

T_1	T_3
	$sum := 0$; read_item(A); $sum := sum + A$; ⋮

read_item(X);
$X := X - N$;
write_item(X);	

	read_item(X);
	$sum := sum + X$;
	read_item(Y);
	$sum := sum + Y$;

read_item(Y);
$Y := Y + N$;
write_item(Y);	

T3 reads X after N is subtracted and reads Y before N is added; a wrong summary is the result (off by N).

Module – 4 Transaction Processing

Why Concurrency Control Is Needed

The Unrepeatable Read Problem

- Another problem that may occur is called unrepeatable read, where a transaction T reads the same item twice and the item is changed by another transaction T′ between the two reads.
- Hence, T receives different values for its two reads of the same item.

Why Recovery Is Needed

Whenever a transaction is submitted to a DBMS for execution, the system is responsible for making sure that either

- All the operations in the transaction are completed successfully and their effect is recorded permanently in the database (committed), **or**
- The transaction does not have any effect on the database or any other transactions(aborted and roll back to old state).

Module – 4 Transaction Processing

Why Concurrency Control Is Needed

Why Recovery Is Needed

- If a transaction fails after executing some of its operations but before executing all of them, then recovery is needed to undone the operations already executed.

Types of Failures

A computer failure (system crash): A hardware, software, or network error occurs in the computer system during transaction execution. Hardware crashes are usually media failures—for example, main memory failure.

A transaction or system error: Some operation in the transaction may cause it to fail, such as integer overflow or **division by zero**. Transaction failure may also occur because of **erroneous parameter values** or because of a logical programming error.

Module – 4 Transaction Processing

Why Concurrency Control Is Needed

Local errors or exception conditions detected by the transaction:

- During transaction execution, certain conditions may occur that necessitate cancellation of the transaction.
- For example, data for the transaction may not be found. An exception condition, such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal, to be canceled.

Concurrency control enforcement:

- The concurrency control may abort a transaction because it violates serializability or it may abort one or more transactions to resolve a state of deadlock among several transactions.
- Transactions aborted because of serializability violations or deadlocks are typically restarted automatically at a later time.

Module – 4 Transaction Processing

Why Concurrency Control Is Needed

Disk failure:

- Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash.
- This may happen during a read or a write operation of the transaction.

Physical problems and catastrophes:

- This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

Module – 4 Transaction Processing

Transaction States and Additional Operations

- A transaction is an atomic unit of work that should either be completed in its entirety or not done at all.

For **recovery purposes**, DBMS needs to keep track of when each transaction;

  - starts,
  - terminates, and
  - commits, or aborts.

Therefore, the recovery manager of the DBMS needs to keep track of the following operations:

- **BEGIN_TRANSACTION**: This marks the beginning of transaction execution.
- **READ or WRITE:** These specify read or write operations on the database items that are executed as part of a transaction.

G C DIVYA, CSE, JIT D

Module – 4 Transaction Processing

Transaction States and Additional Operations

- **END_TRANSACTION:** This specifies that READ and WRITE transaction operations have ended and marks the end of transaction execution.
- **COMMIT_TRANSACTION:** This signals a successful end of the transaction so that any changes (updates) executed by the transaction can be safely committed to the database and will not be undone.
- **ROLLBACK (or ABORT):** This signals that the transaction has ended unsuccessfully, so that any changes or effects that the transaction may have applied to the database must be undone.

Each transaction moves through following execution states:

Active: A transaction goes into an **active state** immediately after it starts execution, where it can execute its READ and WRITE operations.

Module – 4 Transaction Processing

Transaction States and Additional Operations

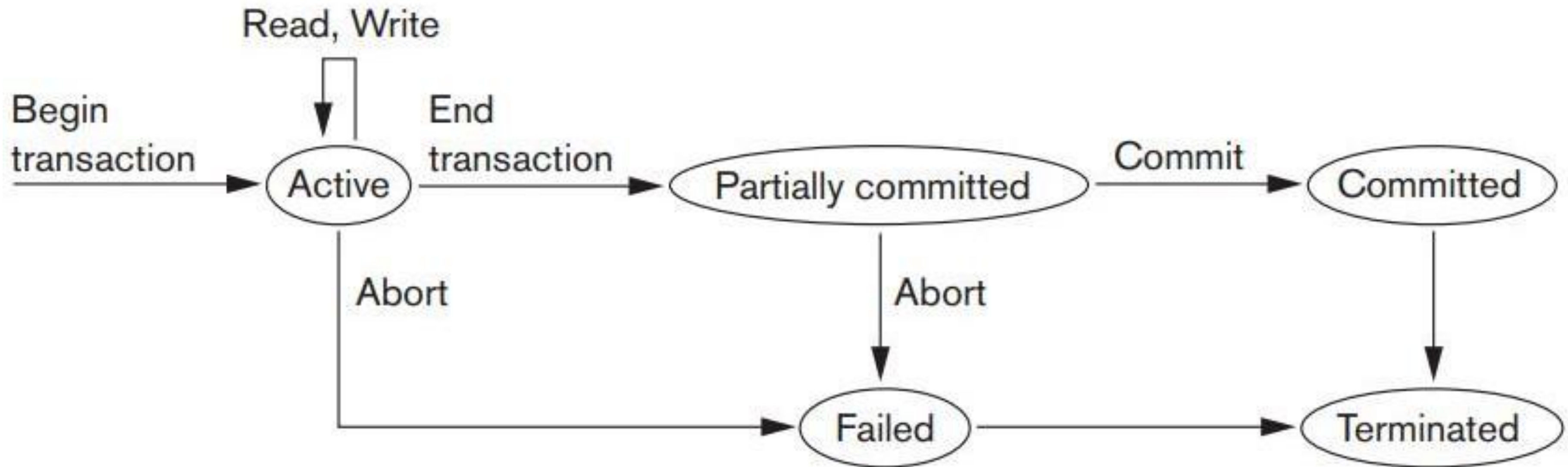


Figure: State transition diagram illustrating the states for transaction execution.

Partially committed: When the transaction ends, it moves to the **partially committed state**. At this point, some types of concurrency control protocols may do additional checks to see if the transaction can be committed or not.

Transaction States and Additional Operations

- Also, some recovery protocols need to ensure that a system failure will not result in an inability to record the changes of the transaction permanently.

Committed:

- If these checks are successful, the transaction is said to have reached its commit point and enters the **committed state**.
- When a transaction is committed, it means execution successfully completed and all its changes must be recorded permanently in the database, even if a system failure occurs.

Failed:

- A transaction can go to the **failed state** if one of the checks fails or if the transaction is aborted during its active state.
- The transaction may then have to be rolled back to undo the effect of its WRITE operations on the database.

Module – 4 Transaction Processing

Transaction States and Additional Operations

Terminated:

- The transaction will reach terminated state either after commited state or failed state.
- Failed or aborted transactions may be restarted later; either automatically or after being resubmitted by the user as brand new transactions.

The System Log

- The system maintains a **log** to keep track of all transaction operations that affect the values of database items, as well as other transaction information that may be useful for recovery from failures.
- Typically, one or more main memory buffers, called the **log buffers**, hold the last part of the log file, so that log entries are first added to the log main memory buffer.

The System Log

- When the log buffer is filled, or when certain other conditions occur, the log buffer is appended to the end of the log file on disk.
- In addition, the log file from disk is periodically backed up to archival storage (tape) to guard against catastrophic failures.

The following are the types of entries—called log records—that are written to the log file and the corresponding action for each log record.

1. **[start_transaction, T]**: Indicates that transaction T has started execution.
2. **[write_item, T, X, old_value, new_value]:** Indicates that transaction T has changed the value of database item X from old_value to new_value.
3. **[read_item, T, X]:** Indicates that transaction T has read the value of database item X.

The System Log

4. **[commit, T]:** Indicates that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
5. **[abort, T]:** Indicates that transaction T has been aborted.

DBMS-Specific Buffer Replacement Policies

- If the required database item for transaction is not available in the buffer and all buffers are occupied from other processes new disk pages are required to be loaded into main memory from disk, a page replacement policy is needed to select the particular buffers to be replaced.

Module – 4 Transaction Processing

DBMS-Specific Buffer Replacement Policies

Some page replacement policies are

Domain Separation (DS) Method:

- In a DBMS, various types of disk pages exist: index pages, data file pages, log file pages, and so on.
- In this method, the DBMS cache(buffer) is divided into separate domains (sets of buffers).
- Each domain handles one type of disk pages, and page replacements within each domain are handled via the basic **LRU** (least recently used) page replacement.

Hot Set Method:

- This page replacement algorithm is useful in queries that have to scan a set of pages repeatedly, such as when a join operation is performed using the nested-loop method.

DBMS-Specific Buffer Replacement Policies

- If the inner loop file is loaded completely into main memory buffers without replacement (the hot set), the join will be performed efficiently because each page in the outer loop file will have to scan all the records in the inner loop file to find join matches.
- The hot set method determines for each database processing algorithm the set of disk pages that will be accessed repeatedly, and it does not replace them until their processing is completed.

The DBMIN Method:

- This page replacement policy uses a model known as **QLSM** (**query locality set model**), which predetermines the pattern of page references for each algorithm for a particular type of database operation.

DBMS-Specific Buffer Replacement Policies

- Depending on the type of access method, the file characteristics, and the algorithm used, the QLSM will estimate the number of main memory buffers needed for each file involved in the operation.
- The DBMIN page replacement policy will calculate a locality set using QLSM for each file instance involved in the query.
- DBMIN then allocates the appropriate number of buffers to each file instance involved in the query based on the locality set for that file instance.

Module – 4 Transaction Processing

Desirable Properties of Transactions

Atomicity:

- The atomicity property requires that we execute a transaction to completion.
- If a transaction fails to complete for some reason, already executed operations of this transaction must undone to old state of database.
- It is the responsibility of the transaction recovery subsystem of a DBMS to ensure atomicity.

Consistency preservation:

- A transaction should be consistency preserving, meaning that if it is completely executed from beginning to end without interference from other transactions, it should take the database from one consistent state to another.

Module – 4 Transaction Processing

Desirable Properties of Transactions

- A **consistent state** of the database satisfies the constraints specified in the schema as well as any other constraints on the database that should hold.
- It is the responsibility of the programmers who write the database application programs and of the DBMS module that enforces integrity constraints.

Isolation:

- The execution of a transaction should not be interfered with by any other transactions executing concurrently.
- The isolation property is enforced by the concurrency control subsystem of the DBMS.

Module – 4 Transaction Processing

Desirable Properties of Transactions

Durability or permanency:

- The changes applied to the database by a committed transaction must persist in the database.
- These changes must not be lost because of any failure.
- The durability property is the responsibility of the recovery subsystem of the DBMS.

Module – 4 Transaction Processing

Schedules (Histories) of Transactions

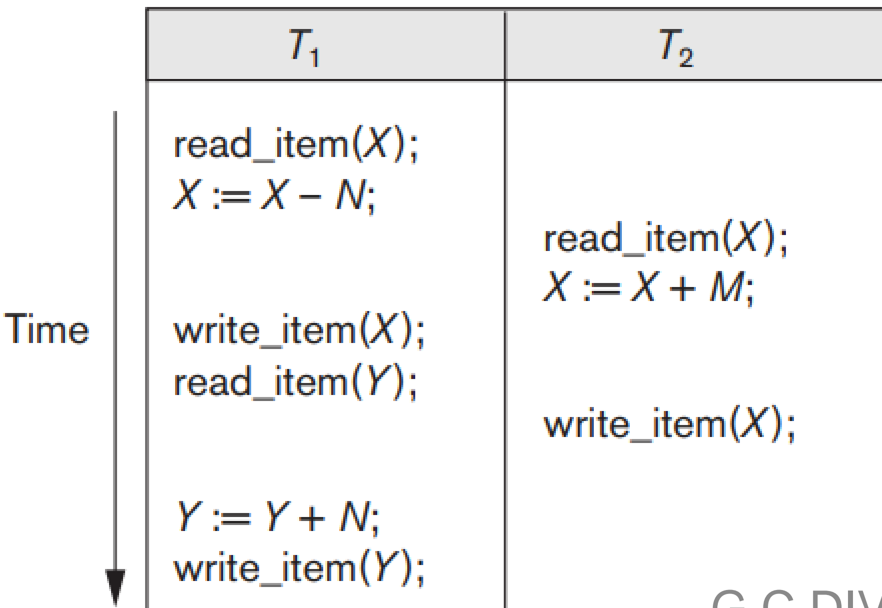
- When transactions are executing concurrently in an interleaved fashion, then the order of execution of operations from all the various transactions is known as a schedule (or history).
- **A schedule** (or history) S of n transactions T_1, T_2, … , T_n is an ordering of the operations of the transactions.
- Operations from different transactions can be interleaved in the schedule S.
- For each transaction T_i that participates in the schedule S, the operations of Ti in S must appear in the same order in which they occur in T_i.
- A shorthand notation for describing a schedule uses the symbols
  - b – begin transaction
  - r – read_item(X)

Module – 4 Transaction Processing

Schedules (Histories) of Transactions

- w – write_item
- e – end transaction
- c – commit
- a – abort and use transaction id as subscript.

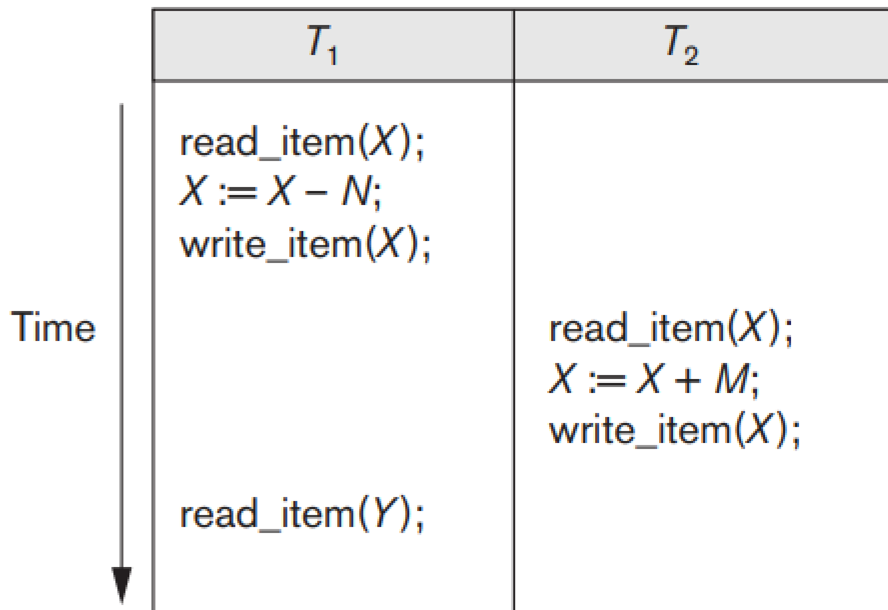
Example



S_a: $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$;

Module – 4 Transaction Processing

Schedules (Histories) of Transactions



S_b: $r_1(X)$; $w_1(X)$; $r_2(X)$; $w_2(X)$; $r_1(Y)$; a_1;

Assuming that transaction T_1 aborted after its read_item(Y) operation:

Module – 4 Transaction Processing

Conflicting Operations in a Schedule

Two operations in a schedule are said to conflict if they satisfy all three of the following conditions:

1. They belong to different transactions
2. They access the same item X; and
3. At least one of the operations is a write_item(X).

Intuitively, two operations are conflicting if changing their order can result in a different outcome compared to previous order of execution.

For example, if we change the order of the two operations $r_1(X)$; $w_2(X)$ to $w_2(X)$; $r_1(X)$ in schedule S_a, then the value of X that is read by transaction T_1 changes, because in the second ordering the value of X is read by $r_1(X)$ after it is changed by $w_2(X)$, whereas in the first ordering the value is read before it is changed. This is called a **read-write conflict**.

Module – 4 Transaction Processing

Conflicting Operations in a Schedule

- If we change the order of two operations such as $w_1(X)$; $w_2(X)$ to $w_2(X)$; $w_1(X)$ in schedule S_a the last value of X will differ because in one case it is written by T_2 and in the other case by T_1. this is called as **write-write conflict**.

Complete schedule

A schedule S of n transactions $T_1, T_2, \ldots, T_n$ is said to be a complete schedule if the following conditions hold:

1. The operations in S are exactly those operations in $T_1, T_2, \ldots, T_n$, including a commit or abort operation as the last operation for each transaction in the schedule.
2. For any pair of operations from the same transaction T_i, their relative order of appearance in S is the same as their order of appearance in T_i.
3. For any two conflicting operations, one of the two must occur before the other in the schedule.

Characterizing Schedules Based on Serializability

If **no interleaving** of operations is permitted, there are only two possible outcomes:

1. Execute all the operations of transaction T1 (in sequence) followed by all the operations of transaction T2 (in sequence).
2. Execute all the operations of transaction T2 (in sequence) followed by all the operations of transaction T1 (in sequence).

These two schedules—called serial schedules.

The concept of **serializability** of schedules is used to identify which schedules are correct when transaction executions have interleaving of their operations in the schedules.

Module – 4 Transaction Processing

Characterizing Schedules Based on Serializability

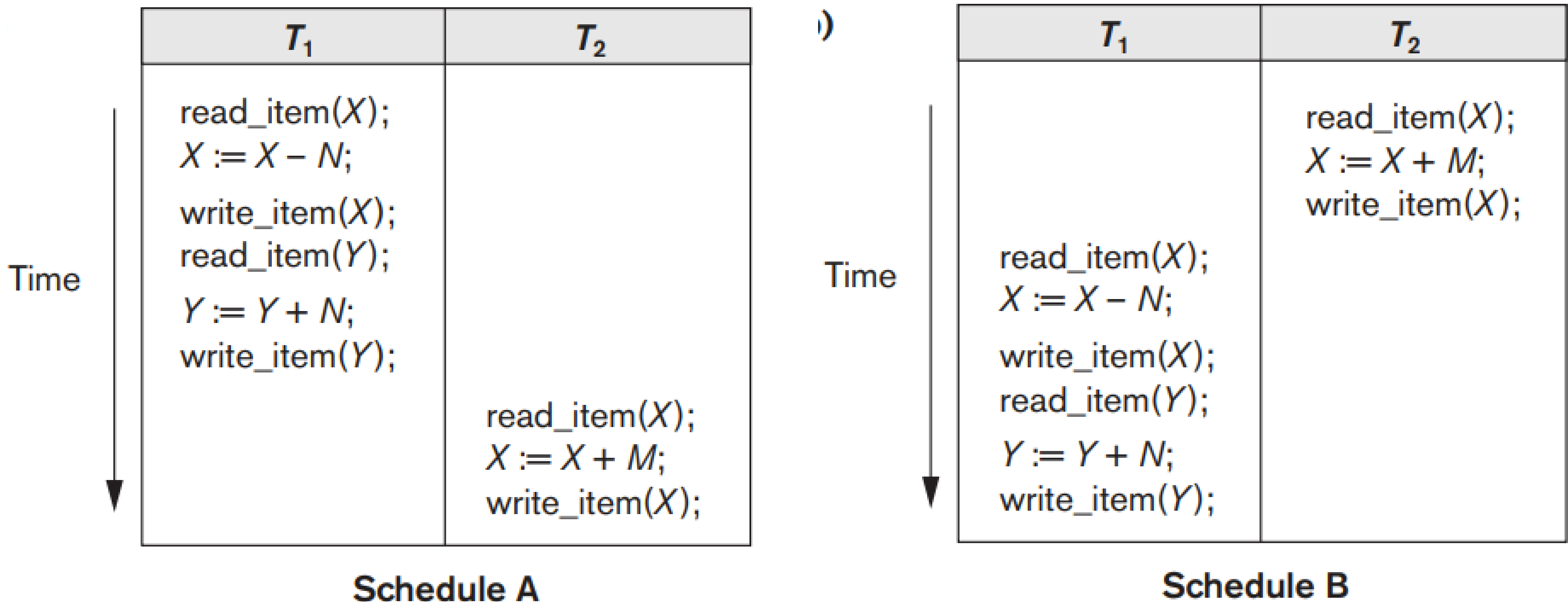


Figure: serial schedules with no interleaving execution.

- Schedules A and B in figures shown above are called serial because the operations of each transaction are executed consecutively, without any interleaved operations from the other transaction.

Module – 4 Transaction Processing

Characterizing Schedules Based on Serializability

- **Formally,** a **schedule S is serial** if, for every transaction T participating in the schedule, all the operations of **T** are **executed consecutively** in the schedule.
- Otherwise, the **schedule is called non-serial**.
- Therefore, in a serial schedule, only one transaction at a time is active, the commit (or abort) of the active transaction initiates execution of the next transaction.

Demerits of serial schedules

- In a serial schedule, if a transaction waits for an I/O operation to complete, we cannot switch the CPU processor to another transaction, thus **wasting valuable CPU processing** time.
- Additionally, if some transaction T is long, the other transactions **must wait for T to complete** all its operations before starting.

Module – 4 Transaction Processing

Characterizing Schedules Based on Serializability

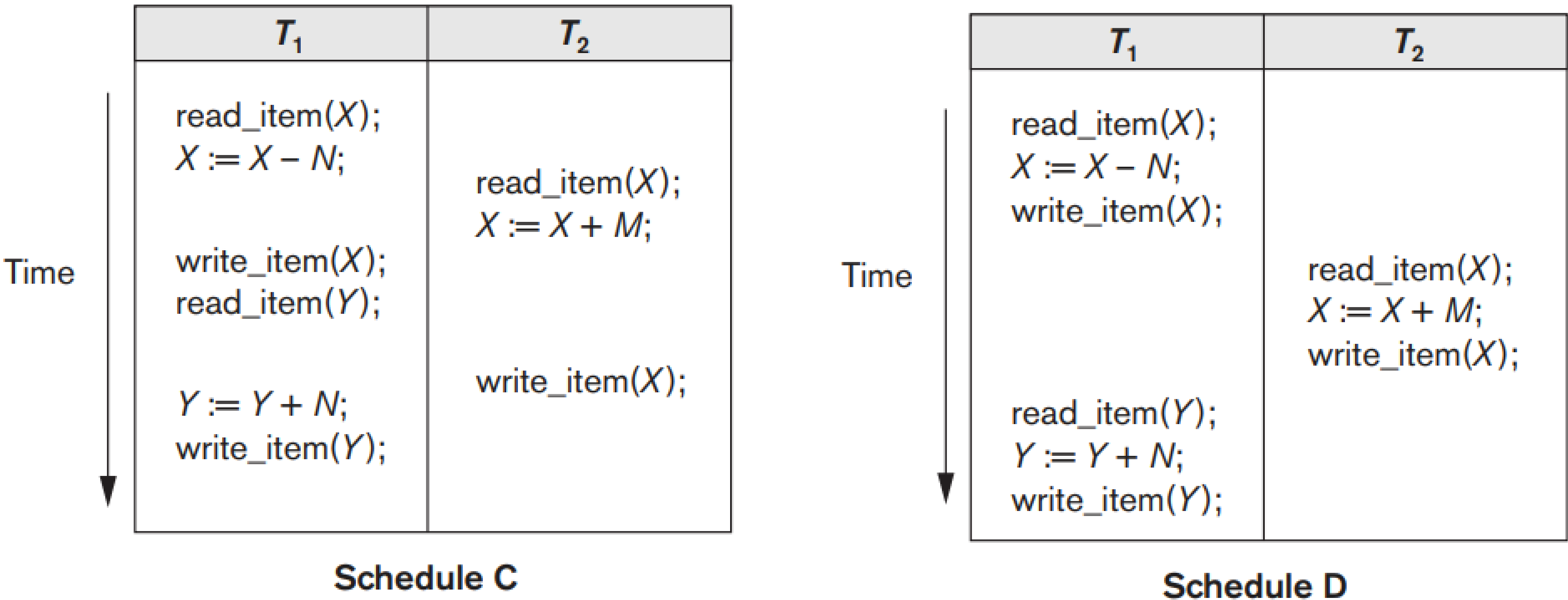


Figure: non serial schedules with interleaving execution.

- Schedule C gives an erroneous result because of the lost update problem.

Characterizing Schedules Based on Serializability

- However, **some non serial schedules give the correct expected result, such as schedule D**(shown in figure above)**.**
- We can determine which of the non serial schedules always give a correct result and which may give erroneous results.
- The concept used to characterize schedules in this manner is that of **serializability of a schedule**.

Module – 4 Transaction Processing

Serializable Schedules

- A schedule **S** of n transactions is **serializable** if it is equivalent to some serial schedule of the same **n** transactions.
- **Note:** there exist n! possible serial schedules of **n** transactions.
- We can **form two disjoint groups of the nonserial schedules**:
  - Those that are equivalent to one (or more) of the serial schedules and hence are serializable.
  - Those that are not equivalent to any serial schedule and hence are not serializable.

When are two schedules considered equivalent?

There are several ways to define schedule equivalence.

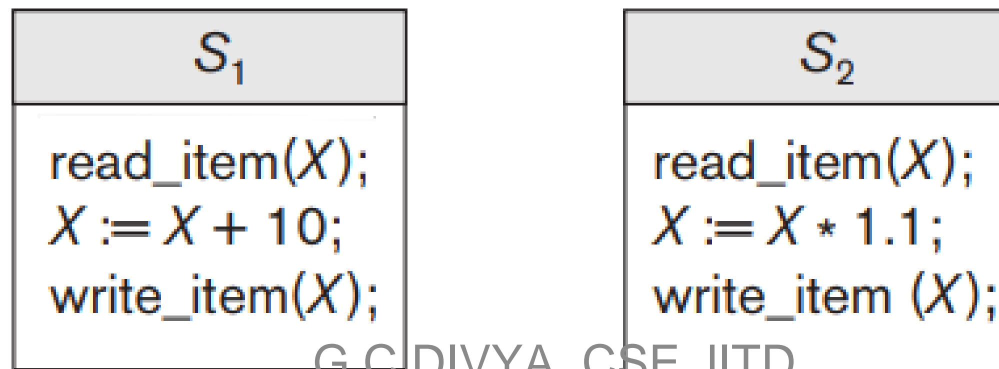
- The **simplest but least satisfactory** definition involves comparing the effects of the schedules on the database.

Module – 4 Transaction Processing

Serializable Schedules

- Two schedules are called **result equivalent** if they produce the same final state of the database.
- Result equivalence alone **cannot be used to define equivalence of schedules**, because two different schedules may accidentally produce the same final state.

For example, schedules S1 and S2 will produce the same final database state if they execute on a database with an initial value of X = 100; however, for **other initial values** of X, the schedules are not result equivalent.



Module – 4 Transaction Processing

Serializable Schedules

- The safest and most general approach to defining schedule equivalence is to focus only on the read_item and write_item operations of the transactions.
- For **two schedules to be equivalent**, the operations applied to each data item affected by the schedules should be applied to that item in both schedules in the same order.
- Two definitions of equivalence of schedules are generally used: **conflict equivalence** and **view equivalence**.
- Two schedules are said to be **conflict equivalent** if the relative order of any two conflicting operations is the same in both schedules.
- If two conflicting operations are applied in different orders in two schedules, the effect can be different on the database or on the transactions in the schedule, and hence the schedules are **not conflict equivalent.**

Source

Text books

1. Fundamentals of Database Systems, Ramez Elmasri and Shamkant B. Navathe, 7th Edition, 2017, Pearson.
2. Database management systems, Ramakrishnan, and Gehrke, 3rd Edition, 2014, McGraw Hill