MODULE – 5

Transaction processing systems are systems with large databases and hundreds of concurrent users executing database transactions. Examples of such systems include airline reservations, banking, credit card processing, online retail purchasing, stock markets, supermarket checkouts, and many other applications. These systems require high availability and fast response time for hundreds of concurrent users. A transaction is used to represent a logical unit of database processing that must be completed in its entirety to ensure correctness. A transaction is typically implemented by a computer program that includes database commands such as retrievals, insertions, deletions, and updates.

5.1 INTRODUCTION TO TRANSACTION PROCESSING

1. Single-User versus Multiuser Systems:

- Classifying a database system according to the number of users who can use the system **concurrently**. A DBMS is -
  - **single-user** if at most one user at a time can use the system, and
  - **multiuser** if many users can use the system and hence access the database concurrently.

- Single-user DBMSs are mostly restricted to personal computer systems; most other DBMSs are multiuser. 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. In multiuser systems, hundreds or thousands of users operate on the database by submitting transactions concurrently to the system.

- Multiple users can access databases and use computer systems simultaneously because of **multiprogramming**, which allows the operating system of the computer to execute multiple programs or **processes** at the same time.
- A single central processing unit (CPU) can only execute at most one process at a time.

- **Multiprogramming operating systems** execute some commands from one process, then suspend that process and execute some commands from the next process, and so on. A process is resumed at the point where it was suspended whenever it gets its turn to use the CPU again. Hence, concurrent execution of processes is actually **interleaved**.

  Figure shows two processes, A and B, executing concurrently in an 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. If the computer system

has multiple hardware processors (CPUs), **parallel processing** of multiple processes is possible, as illustrated by processes C and D in the figure 1.

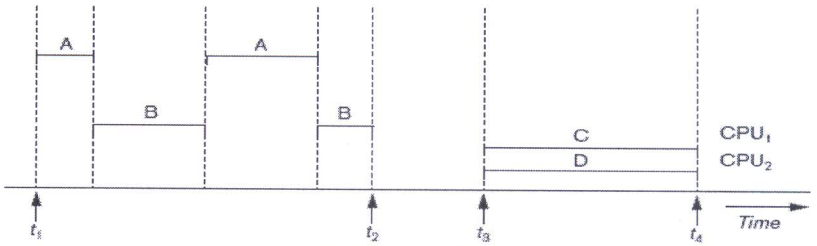


Figure 1: Interleaved processing versus parallel processing of concurrent transactions.

- In a multiuser DBMS, the stored data items are the primary resources that may be accessed concurrently by interactive users or application programs, which are constantly retrieving information from and modifying the database.

2. Transactions, Database Items, Read and Write Operations and DBMS Buffers:

- A **transaction** is an executing program that forms a logical unit of database processing.
- A transaction includes one or more database access operations that 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.
- Transaction boundaries can be specified by explicit **begin transaction** and **end transaction** statements in an application program. Here, 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**.
- 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.
- Each data item has a *unique name* that is not typically used by the programmer. But it is just a means to *uniquely identify each data item.* For example, if the data item granularity is one disk block, then the disk block address can be used as the data item name. If the item granularity is a single record, then the record id can be the item name.

- 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 *named X.*
  - write_**item(*X*):** Writes the value of program variable *X* into the database item named *X.*
- The basic unit of data transfer from disk to main memory is one disk page (disk block).
- 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).

  Step 4 actually updates the database on disk. Sometimes the buffer is not immediately stored to disk, in case additional changes are to be made to the buffer.
- The decision about when to store a modified disk block whose contents are in a main memory buffer is handled by the recovery manager of the DBMS in cooperation with the underlying operating system.
- The DBMS will maintain in the database cache a number of data buffers in main memory.
- Each buffer typically holds the contents of one database disk block, which contains some of the database items being processed. When these buffers are all occupied, and additional database disk blocks must be copied into memory, some buffer replacement policy is used to choose which of the current occupied buffers is to be replaced. Some commonly used buffer replacement policies are LRU (least recently used). If the chosen buffer has been modified, it must be written back to disk before it is reused.
- A transaction includes `read_item` and `write_item` operations to access and update the database. Figure 2 shows examples of two very simple transactions.
- The read-set of a transaction is the set of all items that the transaction reads, and the write-set is the set of all items that the transaction writes. For example, the read-set of T_1 is $\{X, Y\}$ and its write-set is also $\{X, Y\}$.

Figure 2: Transactions T_1 and T_2

- Concurrency control and recovery mechanisms are mainly concerned with the database commands in a transaction. Transactions submitted by the various users may execute concurrently and may access and update the same database items. If this concurrent execution is *uncontrolled*, it may lead to problems, such as an inconsistent database.

3. Why Concurrency Control Is Needed:

Several problems can occur when concurrent transactions execute in an uncontrolled manner.

Consider an 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. Figure 2(a) shows a transaction T_1 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. Figure 2(b) shows a transaction T_2 that just *reserves M* seats on the first flight (X) referenced in transaction T_1.

When a database access program is written, it has the flight number, the flight date, and the number of seats to be booked as parameters; hence, the same program can be used to execute *many different transactions*, each with a different flight number, date, and number of seats to be booked. For concurrency control purposes, a transaction is a *particular execution* of a program on a specific date, flight, and number of seats. In figures 2(a) and (b), the transactions T_1 and T_2 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.

- The types of problems transactions run concurrently are:
  1. **The Lost Update Problem:** It 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. Suppose that transactions T_1 and T_2 are submitted at approximately the same time, and suppose that their operations are interleaved as shown in Figure 3.

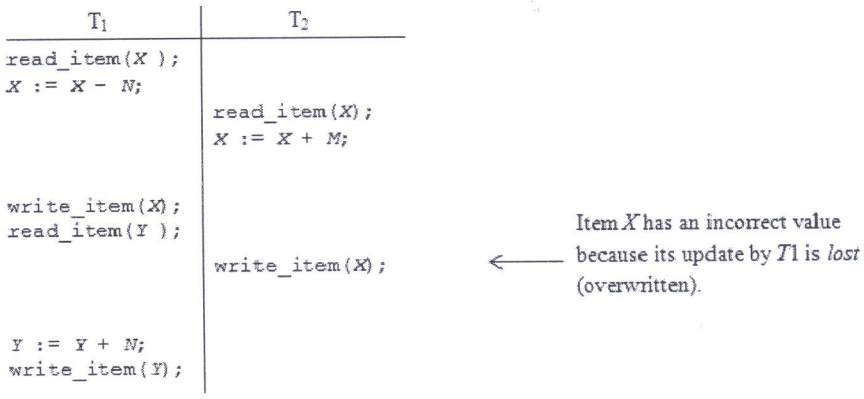


Figure 3: The Lost Update Problem

Then the final value of item X is incorrect because T_2 reads the value of X *before* T_1 changes it in the database, and hence the updated value resulting from T_1 is lost. For example, if $X = 80$ at the start (originally there were 80 reservations on the flight), $N = 5$ (T_1 transfers 5 seat reservations from the flight corresponding to X to the flight corresponding to Y), and $M = 4$ (T_2 reserves 4 seats on X), the final result should be $X = 79$. In the interleaving of operations shown in Figure 3, it is $X = 84$ because the update in T_1 that removed the five seats from X was *lost*.

2. **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. Figure 4 shows an example where T_1 updates item X and then fails before completion, so the system must roll back X to its original value. Before it rolls back, the transaction T_2 reads the *temporary* value of X, which will not be recorded permanently in the database because of the failure of T_1. The value of item X that is read by T_2 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*.

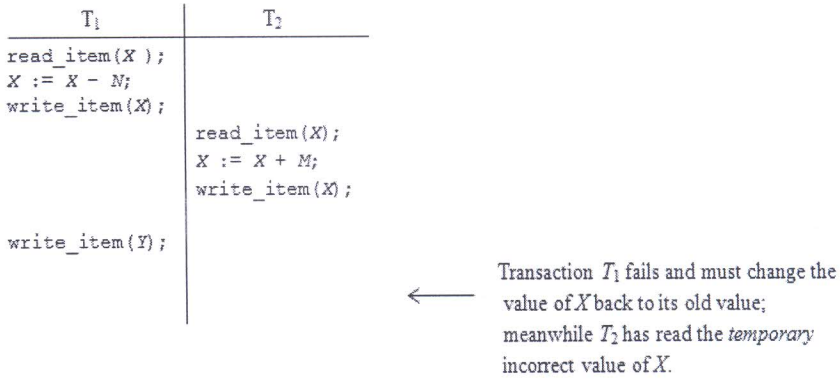


Figure 4: The Temporary Update Problem

3. **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. For example, suppose that a transaction T_3 is calculating the total number of reservations on all the flights; meanwhile, transaction $T1$ is executing. If the interleaving of operations shown in Figure 5 occurs, the result of T_3 will be off by an amount N because T_3 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.

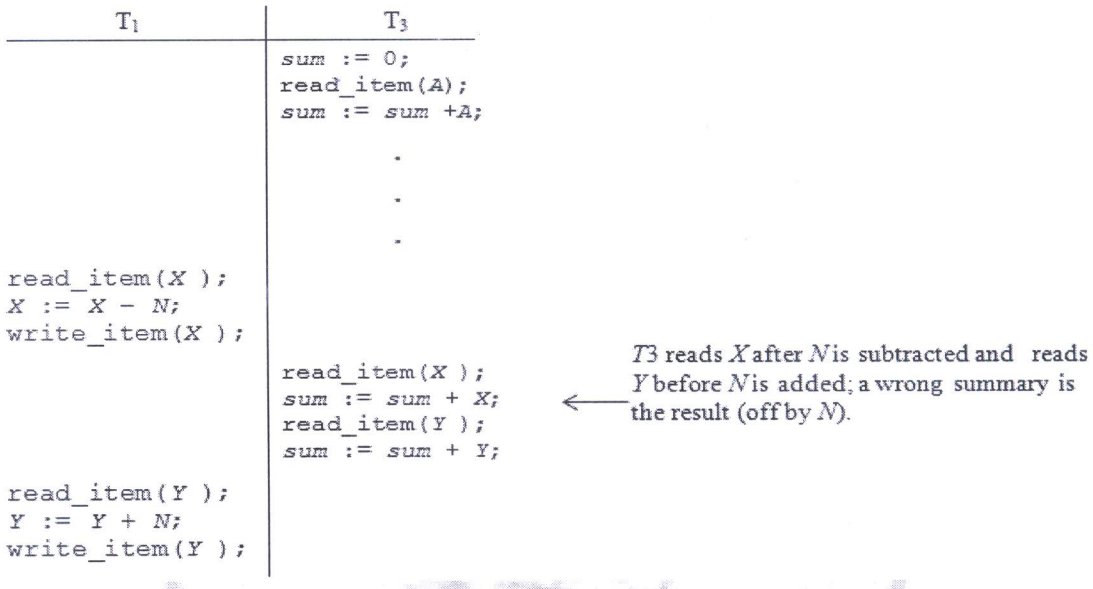


Figure 5: The Incorrect Summary Problem

4. **The Unrepeatable Read Problem:** This occurs 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. This may occur, for example, if during an airline reservation transaction, a customer enquires about seat availability on several flights. When the customer decides on a particular flight, the transaction then reads the number of seats on that flight a second time before completing the reservation, and it may end up reading a different value for the item.

4. 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 (transaction is **committed**), or that the transaction does not have any effect on the database or any other transactions (transaction is **aborted**).

- The DBMS must not permit some operations of a transaction *T* to be applied to the database while other operations of *T* are not, because *the whole transaction* is a logical unit of database processing. If a transaction **fails** after executing some of its operations but before executing all of them, the operations already executed must be undone and have no lasting effect.

- Types of Failures: Failures are generally classified as transaction, system, and media failures. There are several possible reasons for a transaction to fail in the middle of execution:

  1. **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.
  2. **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. Additionally, the user may interrupt the transaction during its execution.
  3. **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. This exception could be programmed in the transaction itself.
  4. **Concurrency control enforcement.** The concurrency control method 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.
  5. **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.
  6. **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. Failures of types 1, 2, 3, and 4 are more common than those of types 5 or 6. Whenever a failure of type 1 through 4 occurs, the system must keep sufficient information to quickly recover from the failure. Disk failure or other catastrophic failures of type 5 or 6 do not happen frequently; if they do occur, recovery is a major task.

5.2 TRANSACTION AND SYSTEM CONCEPTS

1. 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, the system needs to keep track of when each transaction starts, terminates, and commits, or aborts.
- 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.
  - `END_TRANSACTION`: This specifies that `READ` and `WRITE` transaction operations have ended and marks the end of transaction execution. At this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database (committed) or whether the transaction has to be aborted because it violates serializability or for some other reason.
  - `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**.

Figure 6 shows a state transition diagram that illustrates how a transaction moves through its execution states.

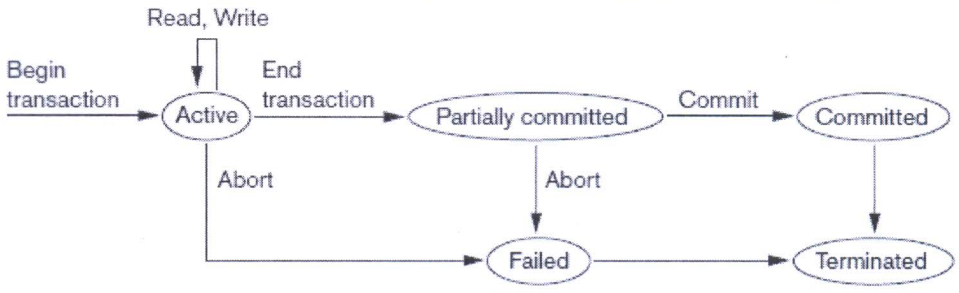


Figure 6: State transition diagram illustrating the states for transaction execution.

- A transaction goes into an **active state** immediately after it starts execution, where it can execute its `READ` and `WRITE` operations.
- 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. 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 (usually by recording changes in the system log). 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 has concluded its execution successfully and all its changes must be recorded permanently in the database, even if a system failure occurs.
- 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.
- The **terminated state** corresponds to the transaction leaving the system.
- The transaction information that is maintained in system tables while the transaction has been running is removed when the transaction terminates.
- Failed or aborted transactions may be *restarted* later either automatically or after being resubmitted by the user as brand new transactions.

2. The System Log:

- To be able to recover from failures that affect transactions, 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 needed to permit recovery from failures.
- The log is a sequential, append-only file that is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure.
- 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.
- 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. In these entries, *T* refers to a unique **transaction-id** that is generated automatically by the system for each transaction and that is used to identify each transaction:

  1. **[`start_transaction, T`]:** Indicates that the 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*.
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.

- Recovery from a transaction failure amounts to either undoing or redoing transaction operations individually from the log.
- If the system crashes, the database can be recovered to a consistent state by examining the log. Because the log contains a record of every WRITE operation that changes the value of some database item, it is possible to **undo** the effect of these WRITE operations of a transaction *T* by tracing backward through the log and resetting all items changed by a WRITE operation of *T* to their *old_values*. **Redo** of an operation may also be necessary if a transaction has its updates recorded in the log but a failure occurs before the sys-on disk from the main memory buffers.

3. Commit Point of a Transaction:

- A transaction *T* reaches its **commit point** when all its operations that access the database have been executed successfully *and* the effect of all the transaction operations on the database have been recorded in the log.
- Beyond the commit point, the transaction is said to be **committed**, and its effect must be *permanently recorded* in the database. The transaction then writes a commit record [commit, *T*] into the log. If a system failure occurs, we can search back in the log for all transactions *T* that have written a [start_transaction, *T*] record into the log but have not written their [commit, *T*] record yet; these transactions may have to be *rolled back* to *undo their effect* on the database during the recovery process.
- Transactions that have written their commit record in the log must also have recorded all their WRITE operations in the log, so their effect on the database can be *redone* from the log records.
- The log file must be kept on disk. Updating a disk file involves copying the appropriate block of the file from disk to a buffer in main memory, updating the buffer in main memory, and copying the buffer to disk. It is common to keep one or more blocks of the log file in main memory buffers, called the **log buffer**, until they are filled with log entries and then to write them back to disk only once, rather than writing to disk every time a log entry is added. This saves the overhead of multiple disk writes of the same log file buffer.
- At the time of a system crash, only the log entries that have been *written back to disk* are considered in the recovery process if the contents of main memory are lost. Hence, *before* a transaction reaches its

commit point, any portion of the log that has not been written to the disk yet must now be written to the disk. This process is called **force-writing** the log buffer to disk before committing a transaction.

4. DBMS-Specific Buffer Replacement Policies:

- The DBMS cache will hold the disk pages that contain information currently being processed in main memory buffers.
- If all the buffers in the DBMS cache are occupied and 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.
- Page replacement policies for database systems:

I. **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 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. Although this achieves better performance on average that basic LRU, it is a *static algorithm,* and so does not adapt to dynamically changing loads because the number of available buffers for each domain is predetermined. Several variations of the DS page replacement policy have been proposed, which add dynamic load-balancing features.

For example, the **GRU** (Group LRU) gives each domain a priority level and selects pages from the lowest-priority level domain first for replacement, whereas another method dynamically changes the number of buffers in each domain based on current workload.

II. **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. 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.

III. **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. 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 (some queries may reference the same file twice, so there would be a locality set for each file instance needed 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. The concept of locality set is analogous to the concept of *working set*, which is used in page replacement policies for processes by the operating system but there are multiple locality sets, one for each file instance in the query.

3. Desirable Properties of Transactions:

- Transactions should possess the **ACID** properties that should be enforced by the concurrency control and recovery methods of the DBMS.
- ACID properties:
  - **Atomicity:** A transaction is an atomic unit of processing. It should either be performed in its entirety or not performed at all.
  - **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.
  - **Isolation:** A transaction should appear as though it is being executed in isolation from other transactions, even though many transactions are executing concurrently. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.
  - **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 *atomicity property* requires that a transaction is executed to completion. It is the responsibility of the *transaction recovery subsystem* of a DBMS to ensure atomicity.
- If a transaction fails to complete for some reason, such as a system crash in the midst of transaction execution, the recovery technique must undo any effects of the transaction on the database. On the other hand, write operations of a committed transaction must be eventually written to disk.
- The preservation of *consistency* is generally considered to be the responsibility of the programmers who write the database programs and of the DBMS module that enforces integrity constraints.
- A **database state** is a collection of all the stored data items (values) in the database at a given point in time. 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.
- A database program should be written in a way that guarantees that, if the database is in a consistent state before executing the transaction, it will be in a consistent state after the *complete* execution of the transaction, assuming that *no interference with other transactions* occurs.

- The *isolation property* is enforced by the *concurrency control subsystem* of the DBMS. If every transaction does not make its updates (write operations) visible to other transactions until it is committed, one form of isolation is enforced that solves the temporary update problem and eliminates cascading but does not eliminate all other problems.
- The *durability property* is the responsibility of the *recovery subsystem* of the DBMS.

Levels of Isolation: A transaction is said to have level 0 (zero) isolation if it does not overwrite the dirty reads of higher-level transactions. Level 1 (one) isolation has no lost updates, and level 2 isolation has no lost updates and no dirty reads. Finally, level 3 isolation (also called *true isolation*) has, in addition to level 2 properties, repeatable reads. Another type of isolation is called **snapshot isolation**, and several practical concurrency control methods are based on this isolation level.

5.4 CHARACTERIZING SCHEDULES BASED ON RECOVERABILITY

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**).

1. Schedules (Histories) of Transactions:

- A **schedule** (or **history**) S of n transactions $T_1, T_2, \ldots, T_n$ 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 T_i in S must appear in the same order in which they occur in T_i. The order of operations in S is considered to be a *total ordering*, meaning *that for any two operations* in the schedule, one must occur before the other. It is possible theoretically to deal with schedules whose operations form *partial orders*, but we will assume for now total ordering of the operations in a schedule.
- A shorthand notation for describing a schedule uses the symbols *b*, *r*, *w*, *e*, *c*, and *a* for the operations `begin_transaction`, `read_item`, `write_item`, `end_transaction`, `commit` and `abort`, respectively, and appends as a *subscript* the transaction id (transaction number) to each operation in the schedule. In this notation, the database item X that is read or written follows the *r* and *w* operations in parentheses.
- The schedule S_a in Figure 3 can be written as follows in this notation:

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

  The schedule for figure 4 called S_b can be written as follows, if we assume that transaction T_1 aborted after its `read_item(Y)` operation:

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

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).

For example, in schedule S_a the operations $r_1(X)$ and $w_2(X)$ conflict, as do the operations $r_2(X)$ and $w_1(X)$, and the operations $w_1(X)$ and $w_2(X)$. But the operations $r_1(X)$ and $r_2(X)$ do not conflict, since they are both read distinct data items X and Y; and the operations $r_1(X)$ and $w_1(X)$ do not conflict because they belong to the same transaction.

- Two operations are conflicting if changing their order can result in a different outcome. For example, if we change the order of the two operations $r_1(X)$; $w_2(X)$ to $w_2(X)$; $r_1(X)$, 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**.
- The other type is called a **write-write conflict** is illustrated by the case where the order of two operations such as $w_1(X)$; $w_2(X)$ is changed to $w_2(X)$; $w_1(X)$. For a write-write conflict, 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.
- Two read operations are not conflicting because changing their order makes no difference in outcome.

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.

The preceding condition (3) allows for two *nonconflicting operations* to occur in the schedule without defining which occurs first, thus leading to the definition of a schedule as a **partial order** of the operations in the n transactions. A total order must be specified in the schedule for any pair of conflicting operations (condition 3) and for any pair of operations from the same transaction (condition 2). Condition 1 simply states that all operations in the transactions must appear in the complete schedule. Since every transaction has either committed or aborted, a complete schedule will *not contain any active transactions* at the end of the schedule.

In general, it is difficult to encounter complete schedules in a transaction processing system because new transactions are continually being submitted to the system. The **committed projection** $C(S)$ of a schedule S includes only the operations in S that belong to committed transactions— that is, transactions T_i whose commit operation c_i is in S.

2. Characterizing Schedules Based on Recoverability:

- Once a transaction T is committed, it should *never* be necessary to roll back T. This ensures that the durability property of transactions is not violated. The schedules that theoretically meet this criterion are called *recoverable schedules*.
- A schedule where a committed transaction may have to be rolled back during recovery is called **nonrecoverable** and hence should not be permitted by the DBMS.
- A schedule S is recoverable if no transaction T in S commits until all transactions T' that have written some item X that T reads have committed.
- A transaction T **reads** from transaction T' in a schedule S if some item X is first written by T' and later read by T. In addition, T' should not have been aborted before T reads item X, and there should be no transactions that write X after T' writes it and before T reads it (unless those transactions, if any, have aborted before T reads X).
- If sufficient information is kept (in the log), a recovery algorithm can be devised for any recoverable schedule.
- The (partial) schedules S_a and S_b from are both recoverable, since they satisfy the above definition. Consider the schedule S_a' given below, which is the same as schedule S_a except that two commit operations have been added to S_a:

  S_a': $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; c_2; $w_1(Y)$; c_1;

  S_a' is recoverable, even though it suffers from the lost update problem; this problem is handled by serializability theory.

  Consider the two (partial) schedules S_c and S_d:

  S_c : $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; c_2; a_1;

  S_d : $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$; c_1; c_2;

  S_e : $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$; a_1; a_2;

  S_c is not recoverable because T_2 reads item X from T_1, but T_2 commits before T_1 commits. The problem occurs if T_1 aborts after the c_2 operation in S_c; then the value of X that T_2 read is no longer valid and T_2 must be aborted *after* it is committed, leading to a schedule that is *not recoverable*. For

the schedule to be recoverable, the c_2 operation in S_c must be postponed until after T_1 commits, as shown in S_d. If T_1 aborts instead of committing, then T_2 should also abort as shown in S_e, because the value of X it read is no longer valid. In S_e, aborting T_2 is acceptable since it has not committed yet, which is not the case for the non-recoverable schedule S_c. In a recoverable schedule, no committed transaction ever needs to be rolled back, and so the definition of a committed transaction as durable is not violated.

- A phenomenon known as **cascading rollback** (or **cascading abort**) occurs in some recoverable schedules, where an *uncommitted* transaction has to be rolled back because it read an item from a transaction that failed. This is illustrated in schedule S_e, where transaction T_2 has to be rolled back because it read item X from T_1, and T_1 then aborted.
- A schedule is said to be **cascadeless**, or to **avoid cascading rollback**, if every transaction in the schedule reads only items that were written by committed transactions. In this case, all items read will not be discarded because the transactions that wrote them have committed, so no cascading rollback will occur. To satisfy this criterion, the $r_2(X)$ command in schedules S_d and S_e must be postponed until after T_1 has committed (or aborted), thus delaying T_2 but ensuring no cascading rollback if T_1 aborts.
- A **strict schedule**, in which transactions can *neither read nor write* an item X until the last transaction that wrote X has committed (or aborted).
- In a strict schedule, the process of undoing a `write_item(X)` operation of an aborted transaction is simply to restore the **before image** (old_value or BFIM) of data item X. This simple procedure always works correctly for strict schedules, but it may not work for recoverable or cascadeless schedules. For example, consider schedule S_f:

  S_f: $w_1(X, 5)$; $w_2(X, 8)$; a_1;

  Suppose that the value of X was originally, which is the before image stored in the system log along with the $w_1(X, 5)$ operation. If T_1 aborts, as in S_f, the recovery procedure that restores the before image of an aborted write operation will restore the value of X to 9, even though it has already been changed to 8 by transaction T_2, thus leading to potentially incorrect results. Although schedule S_f is cascadeless, it is not a strict schedule, since it permits T_2 to write item X even though the transaction T_1 that last wrote X had not yet committed (or aborted). A strict schedule does not have this problem.

- Any strict schedule is also cascadeless, and any cascadeless schedule is also recoverable. Suppose there are i transactions $T_1, T_2, \ldots, T_i$, and their number of operations are $n_1, n_2, \ldots, n_i$, respectively. If a set of *all possible schedules* of these transactions is made, the schedules can be divided into two

disjoint subsets: recoverable and nonrecoverable. The cascadeless schedules will be a subset of the recoverable schedules, and the strict schedules will be a subset of the cascadeless schedules. Thus, all strict schedules are cascadeless, and all cascadeless schedules are recoverable. Most recovery protocols allow only strict schedules, so that the recovery process itself is not complicated.

5.5 CHARACTERIZING SCHEDULES BASED ON SERIALIZABILITY

The types of schedules that are always considered to be *correct* when concurrent transactions are executing are known as *serializable schedules*. Suppose that two users—for example, two airline reservations agents—submit to the DBMS transactions T_1 and T_2 in Figure 2 at approximately the same time. If no interleaving of operations is permitted, there are only two possible outcomes:

1. Execute all the operations of transaction T_1 (in sequence) followed by all the operations of transaction T_2 (in sequence).
2. Execute all the operations of transaction T_2 (in sequence) followed by all the operations of transaction T_1 (in sequence).

These two serial schedules are shown in Figures 7(a) and (b), respectively. If interleaving of operations is allowed, there will be many possible orders in which the system can execute the individual operations of the transactions. Two possible schedules are shown in Figure 7(c). 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.

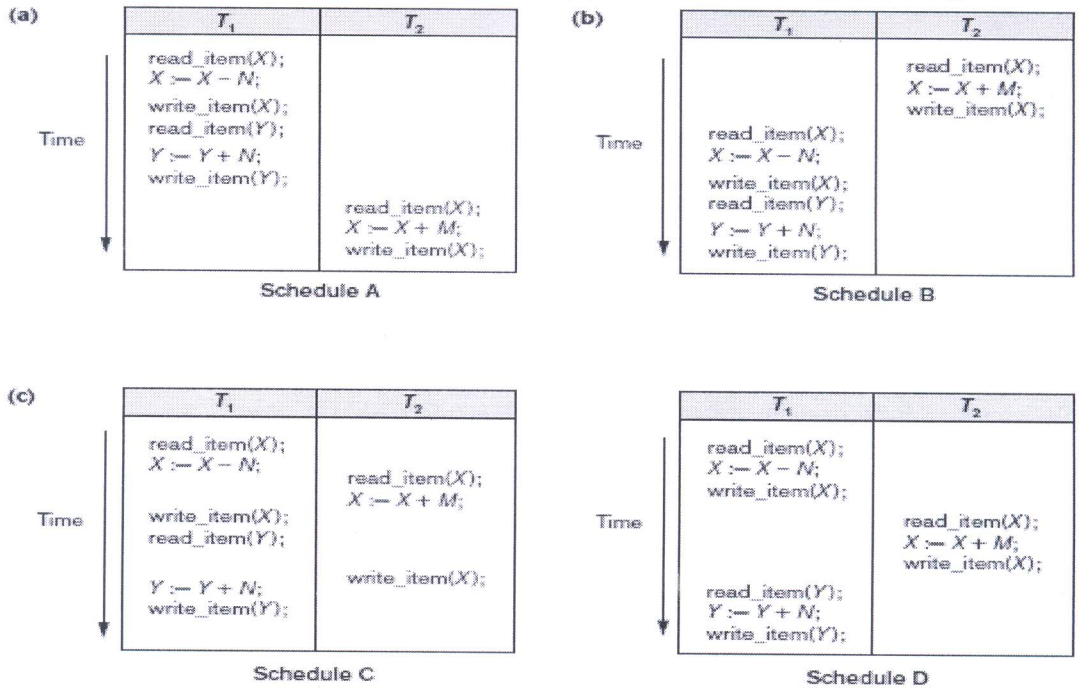


Figure 7

1. Serial, Nonserial, and Conflict-Serializable Schedules:

- Schedules A and B in Figures 7(a) and (b) are called *serial* because the operations of each transaction are executed consecutively, without any interleaved operations from the other transaction. In a serial schedule, entire transactions are performed in serial order: T_1 and then T_2 in Figure 7(a), and T_2 and then T_1 in Figure 7(b).
- Schedules C and D in Figure 7(c) are called *nonserial* because each sequence interleaves operations from the two transactions.
- 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 **nonserial**.
- 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. No interleaving occurs in a serial schedule. Hence, it does not matter which transaction is executed first. As long as every transaction is executed from beginning to end in isolation from the operations of other transactions, a correct end result is obtained.
- Serial schedules limit concurrency by prohibiting interleaving of operations. In a serial schedule, if a transaction waits for an I/O operation to complete, the CPU processor cannot be switched 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. Hence, serial schedules are *unacceptable* in practice. If other schedules are *equivalent* to a serial schedule, these schedules can be allowed to occur.

Assume that the initial values of database items are $X = 90$ and $Y = 90$ and that $N = 3$ and $M = 2$. After executing transactions T_1 and T_2, the database values are expected to be $X = 89$ and $Y = 93$, according to the meaning of the transactions. Executing either of the serial schedules A or B gives the correct results. Consider the nonserial schedules C and D. Schedule C gives the results $X = 92$ and $Y = 93$, in which the X value is erroneous, whereas schedule D gives the correct results.

Schedule C gives an erroneous result because of the *lost update problem*. Transaction T_2 reads the value of X before it is changed by transaction T_1, so only the effect of T_2 on X is reflected in the database. The effect of T_1 on X is *lost*, overwritten by T_2, leading to the incorrect result for item X. But some nonserial schedules give the correct expected result, such as schedule D.

- A schedule S of n transactions is **serializable** if it is *equivalent to some serial schedule* of the same n transactions.

- Two disjoint groups of the nonserial schedules can be formed- those that are equivalent to one (or more) of the serial schedules and hence are serializable, and those that are not equivalent to *any* serial schedule and hence are not serializable.
- Two schedules are called **result equivalent** if they produce the same final state of the database. However, two different schedules may accidentally produce the same final state. For example, in Figure 8, schedules S_1 and S_2 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.

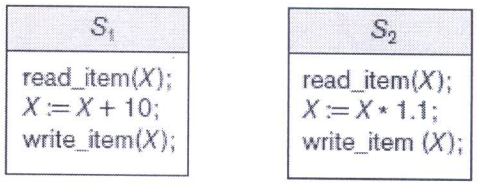


Figure 8: Two schedules that are result equivalent for the initial value of $X = 100$ but are not result equivalent in general.

These schedules execute different transactions, so they definitely should not be considered equivalent. Hence, result equivalence alone cannot be used to define equivalence of schedules.

- 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*.

Conflict Equivalence of Two Schedules: Two schedules are said to be **conflict equivalent** if the relative order of any two *conflicting operations* is the same in both schedules. Two operations in a schedule are said to *conflict* if they belong to different transactions, access the same database item, and either both are `write_item` operations or one is a `write_item` and the other a `read_item`.

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. If a read and write operation occur in the order $r_1(X)$, $w_2(X)$ in schedule S_1, and in the reverse order $w_2(X)$, $r_1(X)$ in schedule S_2, the value read by $r_1(X)$ can be different in the two schedules. Similarly, if two write operations occur in the order $w_1(X)$, $w_2(X)$ in S_1, and in the reverse order $w_2(X)$, $w_1(X)$ in S_2, the next $r(X)$ operation in the two schedules will read potentially different values; or if these are the last operations writing item X in the schedules, the final value of item X in the database will be different.

Serializable Schedules: A schedule S is said to be **serializable** if it is (conflict) equivalent to some serial schedule S'. In such a case, the *nonconflicting* operations in S can be reordered until the equivalent serial

schedule S' is formed. According to this definition, schedule D in Figure 7(d) is equivalent to the serial schedule A in Figure 7(a). In both schedules, the `read_item(X)` of T_2 reads the value of X written by T_1, whereas the other read_item operations read the database values from the initial database state. Additionally, T_1 is the last transaction to write Y, and T_2 is the last transaction to write X in both schedules. Because A is a serial schedule and schedule D is equivalent to A, D is a serializable schedule. The operations $r_1(Y)$ and $w_1(Y)$ of schedule D do not conflict with the operations $r_2(X)$ and $w_2(X)$, since they access different data items. Therefore, we can move $r_1(Y)$, $w_1(Y)$ before $r_2(X)$, $w_2(X)$, leading to the equivalent serial schedule T_1, T_2.

Schedule C in Figure 7(c) is not equivalent to either of the two possible serial schedules A and B, and hence is *not serializable*. Trying to reorder the operations of schedule C to find an equivalent serial schedule fails because $r_2(X)$ and $w_1(X)$ conflict, which means that we cannot move $r_2(X)$ down to get the equivalent serial schedule T_1, T_2. Similarly, because $w_1(X)$ and $w_2(X)$ conflict, we cannot move $w_1(X)$ down to get the equivalent serial schedule T_2, T_1.

2. Testing for Serializability of a Schedule:

The algorithm can be used to test a schedule for conflict serializability. The algorithm looks at only the `read_item` and `write_item` operations in a schedule to construct a **precedence graph** (or **serialization graph**), which is a **directed graph** $G = (N, E)$ that consists of a set of nodes $N = \{T_1, T_2, \ldots, T_n\}$ and a set of directed edges $E = \{e_1, e_2, \ldots, e_m\}$. There is one node in the graph for each transaction T_i in the schedule. Each edge e_i in the graph is of the form $(T_j \rightarrow T_k)$, $1 \leq j \leq n$, $1 \leq k \leq n$, where T_j the **starting node** of e_i and T_k is the **ending node** of e_i. Such an edge from node T_j to node T_k is created by the algorithm if pair of conflicting operations exists in T_j and T_k and the conflicting operation in T_j appears in the schedule *before* the *conflicting operation* in T_k.

Algorithm: Testing Conflict Serializability of a Schedule S

1. For each transaction Ti participating in schedule S, create a node labeled T_i in the precedence graph.

2. For each case in S where T_j executes a read_item(X) after T_i executes a `write_item(X)`, create an edge $(T_i \rightarrow T_j)$ in the precedence graph.

3. For each case in S where T_j executes a `write_item(X)` after T_i executes a `read_item(X)`, create an edge $(T_i \rightarrow T_j)$ in the precedence graph.

4. For each case in S where T_j executes a `write_item(X)` after T_i executes a `write_item(X)`, create an edge $(T_i \rightarrow T_j)$ in the precedence graph.

5. The schedule S is serializable if and only if the precedence graph has no cycles.

The precedence graph is constructed as described in the algorithm. If there is a cycle in the precedence graph, schedule S is not (conflict) serializable; if there is no cycle, S is serializable.

A **cycle** in a directed graph is a **sequence of edges** $C = ((T_j \rightarrow T_k), (T_k \rightarrow T_p), \ldots, (T_i \rightarrow T_j))$ with the property that the starting node of each edge— except the first edge—is the same as the ending node of the previous edge, and the starting node of the first edge is the same as the ending node of the last edge (the sequence starts and ends at the same node).

In the precedence graph, an edge from T_i to T_j means that transaction T_i must come before transaction T_j in any serial schedule that is equivalent to S, because two conflicting operations appear in the schedule in that order. If there is no cycle in the precedence graph, an **equivalent serial schedule** S' is created that is equivalent to S, by ordering the transactions that participate in S as follows: *Whenever an edge exists in the precedence graph from T_i to T_j, T_i must appear before T_j in the equivalent serial schedule S'*. The edges $(T_i \rightarrow T_j)$ in a precedence graph can optionally be labeled by the name(s) of the data item(s) that led to creating the edge. Figure 8 shows such labels on the edges.

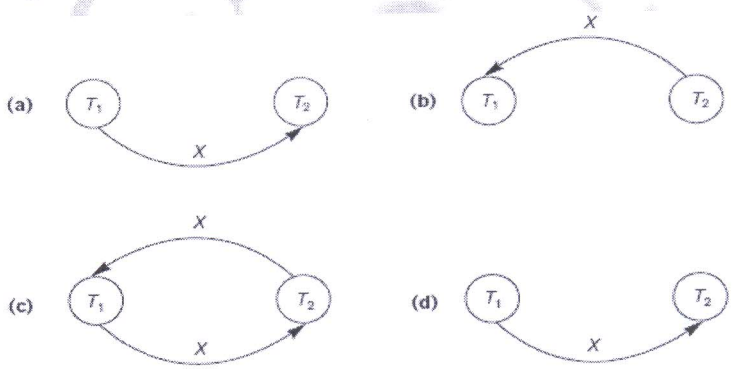


Figure 8: Constructing the precedence graphs for schedules A to D from Figure 7.

When checking for a cycle, the labels are not relevant. In general, several serial schedules can be equivalent to S if the precedence graph for S has no cycle. But if the precedence graph has a cycle, it is easy to show that no equivalent serial schedule can be created, so S is not serializable. The precedence graphs created for schedules A to D, respectively, in Figure 7 appear in Figures 8(a) to (d). The graph for schedule C has a cycle, so it is not serializable. The graph for schedule D has no cycle, so it is serializable, and the equivalent serial schedule is T_1 followed by T_2. The graphs for schedules A and B have no cycles, as expected, because the schedules are serial and hence serializable.

Figure 9 shows the read_item and write_item operations in each transaction. Two schedules E and F for three transactions are also shown.

Transaction T_1
read_item(X);
write_item(X);
read_item(Y);
write_item(Y);

Transaction T_2
read_item(Z);
read_item(Y);
write_item(Y);
read_item(X);
write_item(X);

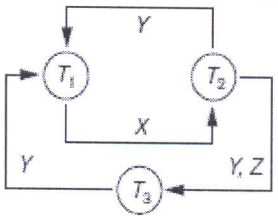
Transaction T_3
read_item(Y);
read_item(Z);
write_item(Y);
write_item(Z);

Figure 9

Transaction T_1	Transaction T_2	Transaction T_3
	read_item(Z);	
	read_item(Y);	
	write_item(Y);	
		read_item(Y);
Time		read_item(Z);
read_item(X);		
write_item(X);		
		write_item(Y);
		write_item(Z);
	read_item(X);	
read_item(Y);		
write_item(Y);		
	write_item(X);	

Schedule E

The precedence graph for schedule E is shown in figure below.



Equivalent serial schedules

None

Reason

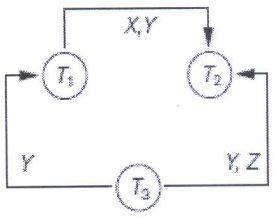
Cycle $X(T_1 \rightarrow T_2), Y(T_2 \rightarrow T_1)$
Cycle $X(T_1 \rightarrow T_2), YZ\ (T_2 \rightarrow T_3), Y(T_3 \rightarrow T_1)$

Schedule E is not serializable because the corresponding precedence graph has cycles.

Transaction T_1	Transaction T_2	Transaction T_3
		read_item(Y);
		read_item(Z);
read_item(X);		
write_item(X);		
		write_item(Y);
		write_item(Z);
Time	read_item(Z);	
read_item(Y);		
write_item(Y);		
	read_item(Y);	
	write_item(Y);	
	read_item(X);	
	write_item(X);	

Schedule F

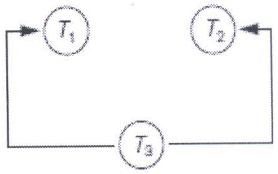
Schedule F is serializable, and the serial schedule equivalent to F is shown in Figure.



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

In general there may be more than one equivalent serial schedule for a serializable schedule. Figure below shows a precedence graph representing a schedule with a node that does not have any incoming edges, and then make sure that the node order for every edge is not violated.



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

$T_3 \rightarrow T_2 \rightarrow T_1$

3. How Serializability Is Used for Concurrency Control:

A serial schedule represents inefficient processing because no interleaving of operations from different transactions is permitted. This can lead to low CPU utilization while a transaction waits for disk I/O, or for a long transaction to delay other transactions, thus slowing down transaction processing considerably. A serializable schedule gives the benefits of concurrent execution without giving up any correctness. In practice, it is difficult to test for the serializability of a schedule. The interleaving of operations from concurrent transactions—which are usually executed as processes by the operating system—is typically determined by the operating system scheduler, which allocates resources to all processes. Factors such as system load, time of transaction submission, and priorities of processes contribute to the ordering of operations in a schedule. Hence, it is difficult to determine how the operations of a schedule will be interleaved beforehand to ensure serializability. If transactions are executed at will and then the resulting schedule is tested for serializability, we must cancel the effect of the schedule if it turns out not to be serializable. This is a serious problem that makes this approach impractical. The approach taken in most commercial DBMSs is to design **protocols** (sets of rules) that—if followed by *every* individual transaction or if enforced by a DBMS concurrency control subsystem—will ensure serializability of *all schedules in which the transactions participate*. Some protocols may allow nonserializable schedules in rare cases to reduce the overhead of the concurrency control method. Another problem is that transactions are submitted continuously to the system, so it is difficult to determine when a schedule begins and when it ends. Serializability theory can be adapted to deal with this problem by considering only the committed

projection of a schedule S. The *committed projection* $C(S)$ of a schedule S includes only the operations in S that belong to committed transactions.

The most common technique, called *two-phase locking*, is based on locking data items to prevent concurrent transactions from interfering with one another, and enforcing an additional condition that guarantees serializability. This is used in some commercial DBMSs. A protocol based on the concept of *snapshot isolation* ensures serializability in most but not all cases; this is used in some commercial DBMSs because it has less overhead than the two-phase locking protocol. Other protocols - *timestamp ordering*, where each transaction is assigned a unique timestamp and the protocol ensures that any conflicting operations are executed in the order of the transaction timestamps; *multiversion protocols*, which are based on maintaining multiple versions of data items; and *optimistic* (also called *certification* or *validation*) *protocols*, which check for possible serializability violations after the transactions terminate but before they are permitted to commit.

4. View Equivalence and View Serializability:

Two schedules S and S' are said to be **view equivalent** if the following three conditions hold:

1. The same set of transactions participates in S and S', and S and S' include the same operations of those transactions.
2. For any operation $r_i(X)$ of T_i in S, if the value of X read by the operation has been written by an operation $w_j(X)$ of T_j (or if it is the original value of X before the schedule started), the same condition must hold for the value of X read by operation $r_i(X)$ of T_i in S'.
3. If the operation $w_k(Y)$ of T_k is the last operation to write item Y in S, then $w_k(Y)$ of T_k must also be the last operation to write item Y in S'.

As long as each read operation of a transaction reads the result of the same write operation in both schedules, the write operations of each transaction must produce the same results. The read operations are hence said to *see the same view* in both schedules. Condition 3 ensures that the final write operation on each data item is the same in both schedules, so the database state should be the same at the end of both schedules. A schedule S is said to be **view serializable** if it is view equivalent to a serial schedule.

❖ **Constrained write assumption** (or **no blind writes**) condition states that any write operation $w_i(X)$ in T_i is preceded by a $r_i(X)$ in T_i and that the value written by $w_i(X)$ in T_i depends only on the value of X read by $r_i(X)$. This assumes that computation of the new value of X is a function $f(X)$ based on the old value of X read from the database. A **blind write** is a write operation in a transaction T on an item X that is not dependent on the old value of X, so it is not preceded by a read of X in the transaction T.

- The definition of view serializability is less restrictive than that of conflict serializability under the **unconstrained write assumption**, where the value written by an operation $w_i(X)$ in T_i can be independent of its old value. This is possible when *blind writes* are allowed, and it is illustrated by the following schedule S_g of three transactions `T1: r1(X); w1(X);` `T2: w2(X);` and `T3: w3(X):`

  S_g: `r1(X); w2(X); w1(X); w3(X); c1; c2; c3;`

  In S_g the operations $w_2(X)$ and $w_3(X)$ are blind writes, since T_2 and T_3 do not read the value of X. The schedule S_g is view serializable, since it is view equivalent to the serial schedule T_1, T_2, T_3. However, S_g is not conflict serializable, since it is not conflict equivalent to any serial.

- Any conflict-serializable schedule is also view serializable but not vice versa. There is an algorithm to test whether a schedule S is view serializable or not. However, the problem of testing for view serializability has been shown to be NP-hard, meaning that finding an efficient polynomial time algorithm for this problem is highly unlikely.

5. Other Types of Equivalence of Schedules:

Some applications can produce schedules that are correct by satisfying conditions less stringent than either conflict serializability or view serializability. An example is the type of transactions known as **debit-credit transactions**—for example, those that apply deposits and withdrawals to a data item whose value is the current balance of a bank account. The semantics of debit-credit operations is that they update the value of a data item X by either subtracting from or adding to the value of the data item. Because addition and subtraction operations are commutative— that is, they can be applied in any order—it is possible to produce correct schedules that are not serializable. For example, consider the following transactions, each of which may be used to transfer an amount of money between two bank accounts:

T_1: `r1(X); X :{equal} X - 10; w1(X); r1(Y); Y :{equal} Y + 10; w1(Y);`

T_2: `r2(Y); Y :{equal} Y - 20; w2(Y); r2(X); X :{equal} X + 20; w2(X);`

Consider the following nonserializable schedule S_h for the two transactions:

S_h: `r1(X); w1(X); r2(Y); w2(Y); r1(Y); w1(Y); r2(X); w2(X);`

With the additional knowledge, or **semantics**, that the operations between each `ri(I)` and `wi(I)` are commutative, the order of executing the sequences consisting of (read, update, write) is not important as long as each (read, update, write) sequence by a particular transaction T_i on a particular item I is not interrupted by conflicting operations. Hence, the schedule S_h is considered to be correct even though it is not serializable.

5.6 TRANSACTION SUPPORT IN SQL

- A transaction is a logical unit of work and is guaranteed to be atomic. A single SQL statement is always considered to be atomic—either it completes execution without an error or it fails and leaves the database unchanged.
- With SQL, there is no explicit `Begin_Transaction` statement. Transaction initiation is done implicitly when particular SQL statements are encountered. But every transaction must have an explicit end statement, which is either a `COMMIT` or a `ROLLBACK`.
- Transaction characteristics are specified by a `SET TRANSACTION` statement in SQL. The characteristics are the *access mode*, the *diagnostic area size*, and the *isolation level*.
- The **access mode** can be specified as `READ ONLY` or `READ WRITE`. The default is `READ WRITE`, unless the isolation level of `READ UNCOMMITTED` is specified (see below), in which case `READ ONLY` is assumed. A mode of `READ WRITE` allows select, update, insert, delete, and create commands to be executed. A mode of `READ ONLY` is for data retrieval.
- The **diagnostic area size** option, `DIAGNOSTIC SIZE` *n*, specifies an integer value *n*, which indicates the number of conditions that can be held simultaneously in the diagnostic area. These conditions supply feedback information (errors or exceptions) to the user or program on the *n* most recently executed SQL statement.
- The **isolation level** option is specified using the statement `ISOLATION LEVEL <isolation>`, where the value for <isolation> can be `READ UNCOMMITTED`, `READ COMMITTED`, `REPEATABLE READ`, or `SERIALIZABLE`. The default isolation level is `SERIALIZABLE`, although some systems use `READ COMMITTED` as their default. The use of the term `SERIALIZABLE` is based on not allowing violations that cause dirty read, unrepeatable read, and phantoms.
- If a transaction executes at a lower isolation level than `SERIALIZABLE`, then one or more of the following three violations may occur:
  1. **Dirty read:** A transaction T_1 may read the update of a transaction T_2, which has not yet committed. If T_2 fails and is aborted, then T_1 would have read a value that does not exist and is incorrect.
  2. **Nonrepeatable read:** A transaction T_1 may read a given value from a table. If another transaction T_2 later updates that value and T_1 reads that value again, T_1 will see a different value.
  3. **Phantoms:** A transaction T_1 may read a set of rows from a table based on some condition specified in the SQL `WHERE`-clause. Now suppose that a transaction T_2 inserts a new row *r* that also satisfies the `WHERE`-clause condition used in T_1, into the table used by T_1. The record *r* is called a **phantom record** because it was not there when T_1 starts but is there when T_1 ends. T_1 may or may not see the

phantom, a row that previously did not exist. If the equivalent serial order is T_1 followed by T_2, then the record r should not be seen; but if it is T_2 followed by T_1, then the phantom record should be in the result given to T_1. If the system cannot ensure the correct behavior, then it does not deal with the phantom record problem.

Table below summarizes the possible violations for the different isolation levels.

	Type of Violation		
Isolation Level	Dirty Read	Nonrepeatable Read	Phantom
READ UNCOMMITTED	Yes	Yes	Yes
READ COMMITTED	No	Yes	Yes
REPEATABLE READ	No	No	Yes
SERIALIZABLE	No	No	No

An entry of *Yes* indicates that a violation is possible and an entry of *No* indicates that it is not possible. `READ UNCOMMITTED` is the most forgiving, and `SERIALIZABLE` is the most restrictive in that it avoids all three of the problems mentioned above.

A sample SQL transaction might look like the following:

```
EXEC SQL WHENEVER SQLERROR GOTO UNDO;
EXEC SQL SET TRANSACTION
READ WRITE
DIAGNOSTIC SIZE 5
ISOLATION LEVEL SERIALIZABLE;
EXEC SQL INSERT INTO EMPLOYEE (Fname, Lname, Ssn, Dno, Salary)
VALUES ('Robert', 'Smith', '991004321', 2, 35000);
EXEC SQL UPDATE EMPLOYEE
SET Salary = Salary * 1.1 WHERE Dno = 2;
EXEC SQL COMMIT;
GOTO THE_END;
UNDO: EXEC SQL ROLLBACK;
THE_END: ... ;
```

The above transaction consists of first inserting a new row in the EMPLOYEE table and then updating the salary of all employees who work in department 2. If an error occurs on any of the SQL statements, the entire transaction is rolled back. This implies that any updated salary (by this transaction) would be restored to its previous value and that the newly inserted row would be removed.

Snapshot Isolation: This isolation level is used in some commercial DBMSs, and some concurrency control protocols exist that are based on this concept. The basic definition of **snapshot isolation** is that a transaction sees the data items that it reads based on the committed values of the items in the *database*

snapshot (or database state) when the transaction starts. Snapshot isolation will ensure that the phantom record problem does not occur, since the database transaction, or in some cases the database statement, will only see the records that were committed in the database at the time the transaction starts. Any insertions, deletions, or updates that occur after the transaction starts will not be seen by the transaction.

QUESTION BANK:

1. Describe the four levels of isolation in SQL. Also discuss the concept of snapshot isolation and its effect on the phantom record problem.
2. Discuss the atomicity, durability, isolation, and consistency preservation properties of a database transaction.
3. What is the system log used for? What are the typical kinds of records in a system log?
4. What are transaction commit points, and why are they important?
5. Draw a state diagram and discuss the typical states that a transaction goes through during execution.
6. What is a schedule (history)? Define the concepts of recoverable, cascadeless and strict schedules.
7. Consider the three transactions T_1, T_2, and T_3, and the schedules S_1 and S_2 given below. Draw the serializability (precedence) graphs for S_1 and S_2, and state whether each schedule is serializable or not. If a schedule is serializable, write down the equivalent serial schedule(s).

   T_1: $r_1(X)$; $r_1(Z)$; $w_1(X)$;

   T_2: $r_2(Z)$; $r_2(Y)$; $w_2(Z)$; $w_2(Y)$;

   T_3: $r_3(X)$; $r_3(Y)$; $w_3(Y)$;

   S_1: $r_1(X)$; $r_2(Z)$; $r_1(Z)$; $r_3(X)$; $r_3(Y)$; $w_1(X)$; $w_3(Y)$; $r_2(Y)$; $w_2(Z)$; $w_2(Y)$;

   S_2: $r_1(X)$; $r_2(Z)$; $r_3(X)$; $r_1(Z)$; $r_2(Y)$; $r_3(Y)$; $w_1(X)$; $w_2(Z)$; $w_3(Y)$; $w_2(Y)$;
8. Consider schedules S3, S4, and S5 below. Determine whether each schedule is strict, cascadeless, recoverable, or nonrecoverable. (Determine the strictest recoverability condition that each schedule satisfies.)

   S_3: $r_1(X)$; $r_2(Z)$; $r_1(Z)$; $r_3(X)$; $r_3(Y)$; $w_1(X)$; c_1; $w_3(Y)$; c_3; $r_2(Y)$; $w_2(Z)$; $w_2(Y)$; c_2;

   S_4: $r_1(X)$; $r_2(Z)$; $r_1(Z)$; $r_3(X)$; $r_3(Y)$; $w_1(X)$; $w_3(Y)$; $r_2(Y)$; $w_2(Z)$; $w_2(Y)$; c_1; c_2; c_3;

   S_5: $r_1(X)$; $r_2(Z)$; $r_3(X)$; $r_1(Z)$; $r_2(Y)$; $r_3(Y)$; $w_1(X)$; c_1; $w_2(Z)$; $w_3(Y)$; $w_2(Y)$; c_3; c_2;
9. Define i) complete schedule ii) conflict equivalent schedule iii) view equivalent schedule.
10. Explain with an algorithm the testing of conflict serializability of a schedule and the importance of precedence graph.
11. Explain transaction support in SQL.
12. Explain the types of problems that occur when transactions run concurrently.
13. Write short notes on page replacement policies for database systems.