CS3CRT06 : Database Management Systems (Core)
Unit V : Transaction Processing and Database Security

Part 1. Transaction Processing, Concurrency Control and Recovery

1. Introduction to Transaction Processing :

Transaction is a logical unit of database processing. Transaction processing systems are systems with large databases and hundreds of concurrent users executing database transactions. Examples: airline reservations, banking, credit card processing, online retail purchasing, stock markets, supermarket checkouts, etc. These systems require high availability and fast response time for hundreds of concurrent users. **The concept of transaction is used to represent a logical unit of database processing that must be completed in its entirety to ensure correctness.**

(i) Single-User versus Multi-user Systems:

Classification of a database system according to the number of users who can use the system **concurrently are Single-User and Multiuser Systems**. A DBMS is a **Single-User System** if at most one user at a time can use the system. A DBMS is a **Multi-User System** if many users can use the system concurrently. For example, an airline reservations system is used by hundreds of travel agents and reservation clerks concurrently. Database systems used in banks, insurance agencies, stock exchanges, supermarkets, and many other applications are multiuser systems.

(ii)Transactions, Database Items, Read and Write Operations, and DBMS Buffers

(a) Transaction : A **transaction** is an executing program that forms a logical unit of database processing. A transaction includes one or more database access operations like insertion, deletion, modification, or retrieval operations. The transaction boundaries are specified by **begin transaction** and **end transaction** statements in an application program. 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**.

(b) Database Item: 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*. It can be a larger unit such as a whole *disk block* or a smaller unit such as an individual *field (attribute) value* of some record in the database. The transaction processing concepts are independent of the data item granularity (size) and apply to data items in general.
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*.

The basic unit of data transfer from disk to main memory is one block.

(c) Read and Write Operations:

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

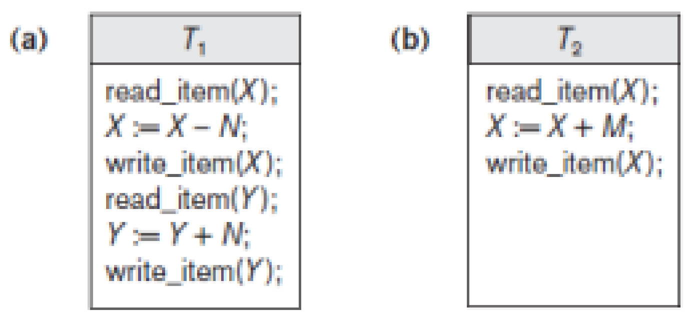
1. Find the address of the disk block that contains item *X*.
2. Copy that disk block into a buffer in main memory
3. Copy item *X* from the buffer to the program variable named *X*.

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 .
3. Copy item *X* from the program variable named *X* into its correct location in the buffer.
4. Store the updated block from the buffer back to disk.

It is step 4 that actually updates the database on disk. In some cases the buffer is not immediately stored to disk, in case additional changes are to be made to the buffer.

A transaction includes read_item and write_item operations to access and update the database. Figure shows examples of two very simple transactions.



(a)

T_1

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

(b)

T_2

read_item(X);
 $X := X + M$;
 write_item(X);

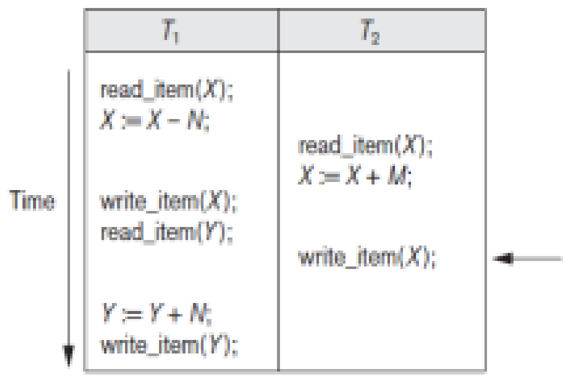
Figure
Two sample transactions. (a) Transaction T_1. (b) Transaction T_2.

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 in Figure is {*X*, *Y*} and its write-set is also {*X*, *Y*}.

(d) DBMS Buffers :
Each data buffer holds the contents of one database disk block, which contains some of the database items being processed. The recovery manager of the DBMS handles the decision about when to store a modified disk block. The number of **data buffers** in main memory are maintained in the **database cache** of the DBMS. 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 buffers is to be replaced. If the chosen buffer has been modified, it must be written back to disk before it is reused.

(iii). Need for Concurrency Control:
Concurrency means simultaneous access to a database. 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.
Several problems can occur when concurrent transactions execute in an uncontrolled manner. Consider the example of a simplified airline reservations database in which a record is stored for each airline flight. ach record includes the number of reserved seats on that flight as a data item and other information.

Figure (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 (b) shows a simpler transaction *T*2 that just *reserves M* seats on the first flight. A database access program can be used to execute *many different transactions*, each with a different flight number, date, and number of seats to be booked.
If two simple transactions T1 and T2 are run concurrently, we may encounter 4 types of problems as follows:
(1) The Lost Update Problem. This problem occurs when two transactions that access the same database items have their operations interleaved and 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 the following Figure:


Time
Item X has an incorrect value because its update by T_1 is *lost* (overwritten).

T_1	T_2
read_item(X); $X := X - N$;	
	read_item(X); $X := X + M$;

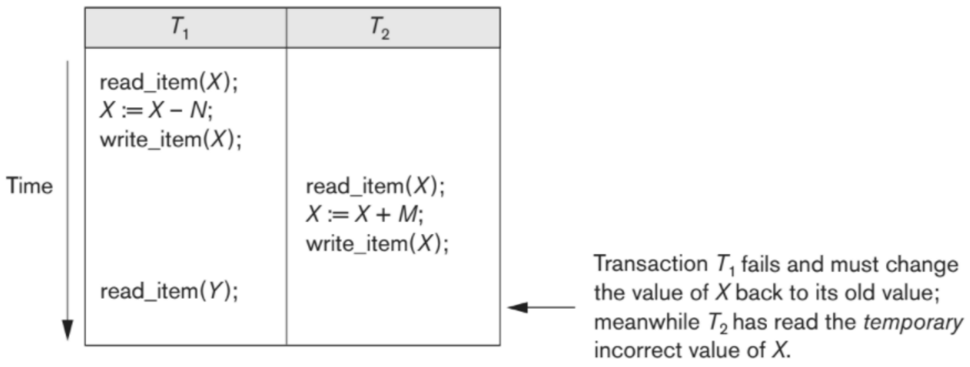
write_item(X);
 read_item(Y);	

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

problem that occur when concurrent execution is uncontrolled. (a) The lost update problem.

The final value of item *X* is incorrect because *T*2 reads the value of *X before T*1 changes it in the database. Hence the updated value of X by *T*1 is lost. For example, if *X* = 80 at the start, *N* = 5 and *M* = 4 , the final result should be *X* = 79. But here 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. The updated item is accessed (read) by another transaction before it is changed back to its original value.



T_1	T_2

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

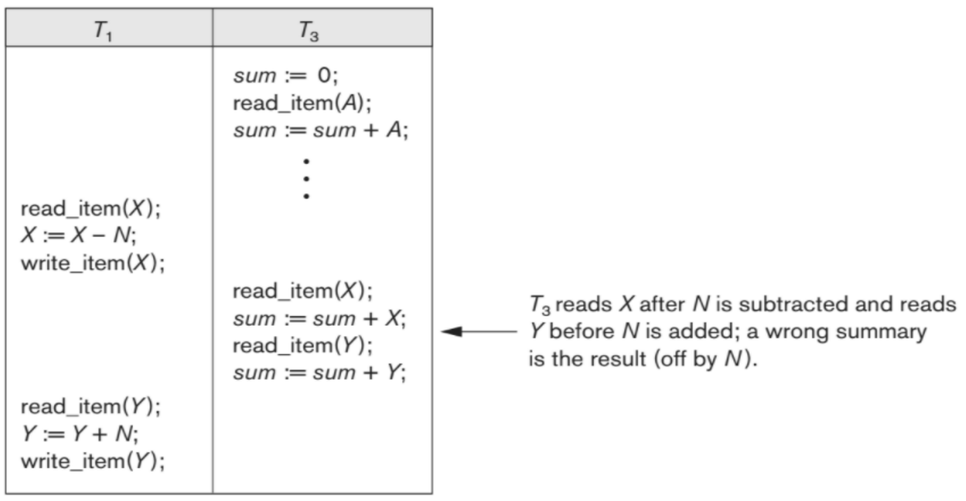
	read_item(X);
	$X := X + M$;
	write_item(X);

read_item(Y);	

Transaction T_1 fails and must change the value of X back to its old value; meanwhile T_2 has read the *temporary* incorrect value of X.

The Figure above shows an example where *T*1 updates item *X* and then fails before completion, so the system must change *X* back to its original value. But before it can do so, 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. This problem is also known as the *dirty read 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.



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);	

T_3 reads X after N is subtracted and reads Y before N is added; a wrong summary is the result (off by N).

For example, suppose that a transaction *T*3 is calculating the total number of reservations on all the flights; meanwhile, transaction *T*1 is executing. If the interleaving of operations shown in the above Figure occurs, the result of *T*3 will be wrong 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.

(4) 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. This may occur, for example, if during an airline reservation transaction, a customer inquires 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.

(iv). Need for Recovery:

- 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, or that the transaction does nothing on the database or any other transactions.

In the first case, the transaction is **committed**. In the second case, the transaction is **aborted**. If a transaction **fails** after executing some of its operations but before executing all of them, the operations already executed must be undone.

Types of Failures.
Failures are generally classified as transaction, system, and media failures.

The possible reasons for a transaction to fail in the middle of execution are:

- **A computer failure (system crash).** A hardware, software, or network error occurs in the computer system during transaction execution is called **system crash**. Hardware crashes are usually media failures Eg: main memory failure.
- **A transaction or system error.** Some operations in the transaction such as integer overflow or division by zero may cause it to fail. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error. Sometimes, the user may interrupt the transaction during its execution.
- **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 cancelled. This exception could be programmed in the transaction itself, and in such a case would not be considered as a transaction failure.

2. Transaction and System Concepts:
(i). 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. 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.
■ 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. Any changes (updates) executed by the transaction will not be undone.
■ ROLLBACK (or ABORT). This signals that the transaction has *ended unsuccessfully. A*ny changes made by the transaction must be **undone**.

State transition diagram illustrating the states for transaction execution.

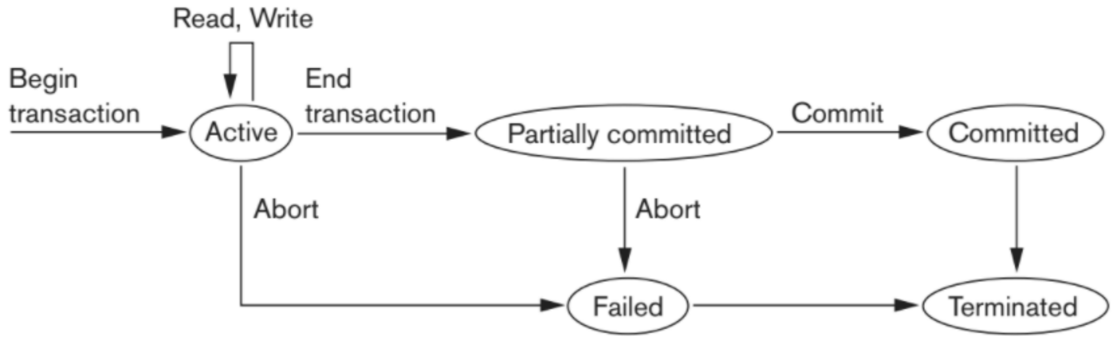


Figure shows a state transition diagram that illustrates how a transaction moves through its execution

states.
Active state: A transaction goes into an **active state** immediately after it starts execution. It can execute its READ and WRITE operations in this state.
Partially committed state: When the transaction ends, it moves to the **partially committed state**. At this point, 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 state: Once this check is 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.
Failed state: However, 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.
Terminated state: The **terminated state** corresponds to the transaction leaving the system. The transaction information that is maintained in system tables is removed when the transaction terminates. Failed or aborted transactions may be *restarted* later either automatically or after being resubmitted by the user as new transactions.

(ii). The System Log:
The system maintains a **log** to enable recovery from failures that affect transactions**.** The log keeps track of all transaction operations that affect the values of database items, as well as other transactions. This log information may be needed for the 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 (sudden and total failure).
Usually, a log file is stored on *disk.* One (or more) main memory buffers hold the last part of the log file. The new log entries are first added to the 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*. The log file from disk is periodically backed up to archival storage (tape) to guard against catastrophic failures.
Suppose *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. The **log records** that are written to the log file are:
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*.
4. **[commit, *T*]**. Indicates that transaction *T* has completed successfully.
5. **[abort, *T*]**. Indicates that transaction *T* has been aborted.

All permanent changes to the database occur within transactions.

Recovery from a transaction failure means either undoing or redoing transaction operations individually from the log. If the system crashes, we can recover to a consistent database state by examining the log.
Undo transaction operations: The log contains a record of every WRITE operation that changes the value of some database item. So, it is possible to undo the effect of these WRITE operations of a transaction T by tracing backward through the log. Then reset all items changed by a WRITE operation of T to their old_values.
Redo transaction operations: Redo of an operation may also be necessary if a transaction has its updates recorded in the log but a failure occurs before the system can write these new values to the actual database on disk from the main memory buffers.

(iii) Commit Point of a Transaction
When a transaction is **committed**, its effect is permanently recorded in the database. The transaction then writes a commit record [commit, T] into the log. 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.
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 **redo**ne 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. So 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 before committing a transaction.

4. Desirable Properties of Transactions :

Transactions should possess several properties, called the **ACID** properties.They should be enforced by the concurrency control and recovery methods of the DBMS. The following are the ACID properties:
■ **Atomicity.** A transaction is an atomic unit of processing. It should either be performed in its entirety or not performed at all. The atomicity property requires that we execute a transaction to completion. It is the responsibility of the transaction recovery subsystem of a DBMS. If a transaction fails to complete for some reason, such as a system crash in the mid of transaction execution, the recovery technique must undo any effects of that transaction on the database. The write operations of a committed transaction must be written to disk

■ **Consistency preservation.** A transaction should be consistency preserving. It means 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. It is the responsibility of the programmers who write the database programs or of the DBMS module that enforces integrity constraints. 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.
■ **Isolation.** A transaction should appear as though it is being executed in isolation from other transactions, even though many transactions are executing concurrently. The execution of a transaction should not be interfered with any other transactions executing concurrently. The isolation property is enforced by the concurrency control subsystem of the DBMS. There are different **levels of isolation** of a transaction. 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. Level 3 isolation (also called true isolation) has, in addition to level 2 properties, repeatable reads.
■ **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.

Part 2 : Database Security