Unit IV Transaction Management & Concurrency Control

- Contents
- ACID properties
- Serializability and concurrency control
- Lock based concurrency control (2PL, Deadlocks)
- Time stamping methods
- Optimistic methods
- Database recovery management.

- Recommended Books
- Database System and Concepts A Silberschatz, H Korth, S Sudarshan McGrawHill
- Database Systems Rob Coronel Cengage Learning Twelfth Edition
- Programming with PL/SQL for Beginners H. Dand, R. Patil and T. Sambare
- Introduction to Database System C.J.Date

Unit IV	Pre-requisites	Sem. II	Sem. III	Sem. IV	Sem. V	Sem. VI
Transaction Management & Concurrency Control	-	WP,OOP	-	CJ	EJ,AWP,Project	Project

ACID PROPERTIES

To ensure integrity of the data, we require that the database system maintain the following properties of the transactions :

- **Atomicity.** Either all operations of the transaction are reflected properly in the database, or none are.
- **Consistency.** Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database.
- **Isolation.** Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions T_i, and T_j, it appears to T_i, that either T_j finished execution before T_i started, or T_i, started execution after T_i, finished. Thus, each transaction is unaware of other transactions executing concurrently in the system.
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

These properties are often called the ACID properties, the acronym is derived from the first letter of each of the four properties.

To gain a better understanding of ACID properties and the need for them, consider a simplified banking system consisting of several accounts and a set of transactions that access and update those accounts. For the time being, we assume that the database permanently resides on disk, but that some portion of it is temporarily residing in main memory.

Source: https://www.youtube.com/watch?v=dZc6CP-x2d0

SERIALIZABILITY & CONCURRENCY CONTROL

The database system must control concurrent execution of transactions, to ensure that the database state remains consistent. Before we examine how the database system can carry out this task, we must first understand which schedules will ensure consistency, and which schedules will not.

T_1	T_2
read(A)	
A := A − 50	
	read(A)
	temp := A * 0.1
	A := A − temp
	write(A)
	read(B)
write(A)	
read(B)	
B := B + 50	
write(B)	

	B := B + temp
	write(B)

Fig. 1 : Schedule 4-n a concurrent schedule.

Since transactions are programs, it is computationally difficult to determine exactly what operations a transaction performs and how operations of various transactions interact. For this reason, we shall not interpret the type of operations that a transaction can perform on a data item. Instead, we consider only two operations: read and write. We thus assume that, between a read(Q) instruction and a write (Q) instruction on a data item Q, a transaction may perform an arbitrary sequence of operations on the copy of Q that is residing in the local buffer of the transaction. Thus, the only significant operations of a transaction, from a scheduling point of view, are instructions in schedules, as we do in schedule 3 in Figure 1.

In this section, we discuss different forms of schedule equivalence; they lead to the notions of conflict serializability and view serializability.

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Fig. 2 : Schedule 3—showing only the read and write instructions.

LOCK BASED CONCURRENCY CONTROL

A DBMS must be able to ensure that only serializable, recoverable schedules are allowed, and that no actions of committed transactions are lost while undoing aborted transactions. A DBMS typically uses a locking protocol to achieve this. A locking protocol is a set of rules to be followed by each transaction (and enforced by the DBMS), in order to ensure that even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions in some serial order.

Strict Two-Phase Locking (Strict 2PL)

The most widely used locking protocol, called Strict Two-Phase Locking, or Strict 2PL, has two rules. The first rule is

(i) If a transaction T wants to read (respectively, modify) an object, it first requests a shared (respectively, exclusive) lock on the object.
Of course, a transaction that has an exclusive lock can also read the object; an additional shared lock is not required. A transaction that requests a lock is suspended until the DBMS is able to grant it the requested lock. The DBMS keeps track of the locks it has granted and ensures that if a transaction holds an exclusive lock on an object, no other transaction holds a shared or exclusive lock on the same object.

The second rule in Strict 2PL is:

(ii) All locks held by a transaction are released when the transaction is completed.

Requests to acquire and release locks can be automatically inserted into transactions by the DBMS; users need not worry about these details.

In effect the locking protocol allows only 'safe' interleavings of transactions. If two transactions access completely independent parts of the database, they will be able to concurrently obtain the locks that they need and proceed merrily on their ways. On the other hand, if two transactions access the same object, and one of them wants to modify it, their actions are effectively ordered serially—all actions of one of these transactions (the one that gets the lock on the common object first) are completed before (this lock is released and) the other transaction can proceed.

We denote the action of a transaction T requesting a shared (respectively, exclusive) lock on object O as S_T (O) (respectively, X(O)), and omit the subscript denoting the transaction when it is clear from the context. As an example, consider the schedule shown in figure 2. This interleaving could result in a state that cannot result from any serial execution of the three transactions. For instance, T1 could change A from 10 to 20, then T2 (which reads the value 20 for A) could change B from 100 to 200, and then T1 would read the value 200 for B. If run serially, either T1 or T2 would execute first, and read the values 10 for A and 100 for B: Clearly, the interleaved execution is not equivalent to either serial execution.

If the Strict 2PL protocol is used, the above interleaving is disallowed. Let us see why. Assuming that the transactions proceed at the same relative speed as before, T1 would obtain an exclusive lock on A first and then read and write A (Figure 3). Then, T2 would request a lock on A. However, this request cannot be granted until

T1	T2
X(A)	
R(A)	
W(A)	

Fig. 3 : Schedule Illustrating Strict 2PL

T1 releases its exclusive lock on A, and the DBMS therefore suspends T2. T1now proceeds to obtain an exclusive lock on B, reads and writes B, then finally commits, at which time its locks are released. T2's lock request is now granted, and it proceeds. In this example the locking protocol results in a serial execution of the two transactions. In general, however, the actions of different transactions could be interleaved. As an example, consider the interleaving of two transactions shown in Figure 18.6, which is permitted by the Strict 2PL protocol.

Source: https://www.youtube.com/watch?v=xWV1z5Du8N0

Deadlock

In a multi-process system, deadlock is an unwanted situation that arises in a shared resource environment, where a process indefinitely waits for a resource that is held by another process.

For example, assume a set of transactions $\{T_0, T_1, T_2, ...,T_n\}$. T_0 needs a resource X to complete its task. Resource X is held by T_1, and T_1 is waiting for a resource Y, which is held by T_2. T_2 is waiting for resource Z, which is held by T_0. Thus, all the processes wait for each other to release resources. In this situation, none of the processes can finish their task. This situation is known as a deadlock.

Deadlocks are not healthy for a system. In case a system is stuck in a deadlock, the transactions involved in the deadlock are either rolled back or restarted.

Deadlock Prevention

To prevent any deadlock situation in the system, the DBMS aggressively inspects all the operations, where transactions are about to execute. The DBMS inspects the operations and analyzes if they can create a deadlock situation. If it finds that a deadlock situation might occur, then that transaction is never allowed to be executed.

There are deadlock prevention schemes that use timestamp ordering mechanism of transactions in order to predetermine a deadlock situation.

Wait-Die Scheme

In this scheme, if a transaction requests to lock a resource (data item), which is already held with a conflicting lock by another transaction, then one of the two possibilities may occur −

- If $TS(T_i) < TS(T_j)$ − that is T_i, which is requesting a conflicting lock, is older than T_j − then T_i is allowed to wait until the data-item is available.
- If $TS(T_i) > TS(t_j)$ − that is T_i is younger than T_j − then T_i dies. T_i is restarted later with a random delay but with the same timestamp.

This scheme allows the older transaction to wait but kills the younger one.

Wound-Wait Scheme

In this scheme, if a transaction requests to lock a resource (data item), which is already held with conflicting lock by some another transaction, one of the two possibilities may occur −

- If $TS(T_i) < TS(T_j)$, then T_i forces T_j to be rolled back − that is T_i wounds T_j. T_j is restarted later with a random delay but with the same timestamp.
- If $TS(T_i) > TS(T_j)$, then T_i is forced to wait until the resource is available.

This scheme, allows the younger transaction to wait; but when an older transaction requests an item held by a younger one, the older transaction forces the younger one to abort and release the item.

In both the cases, the transaction that enters the system at a later stage is aborted.

Deadlock Detection

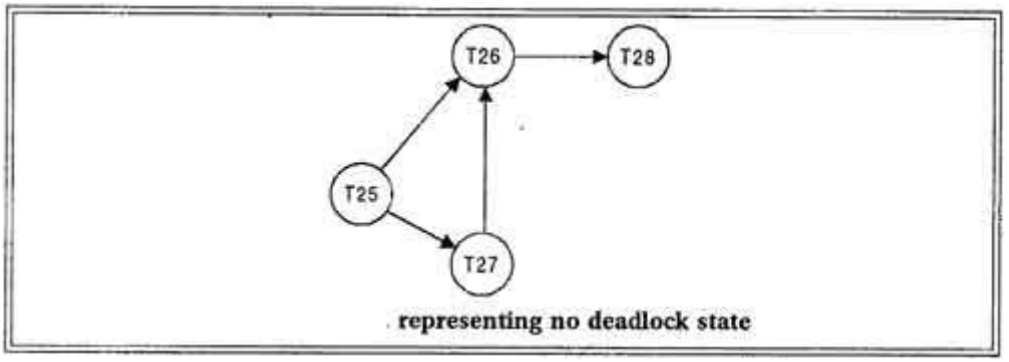
A simple way to detect a state of deadlock is with the help of wait-for graph. This graph is constructed and maintained by the system. One node is created in the wait-for graph for each transaction that is currently executing. Whenever a transaction Ti is waiting to lock an item X that is currently locked by a transaction Tj, a directed edge (Ti->Tj).is created in the wait-for graph. When Tj releases the lock(s) on the items that Ti was waiting for, the directed edge is dropped from the wait-for graph. We have a state of deadlock if and only if the wait-for graph has a cycle. Then each transaction involved in the cycle is said to be deadlocked. To detect deadlocks, the system needs to maintain the wait for graph, and periodically to invoke an algorithm that searches for a cycle in the graph.

To illustrate these concepts, consider the following wait-for graph in figure. Here:

Transaction T25 is waiting for transactions T26 and T27.

Transactions T27 is waiting for transaction T26.

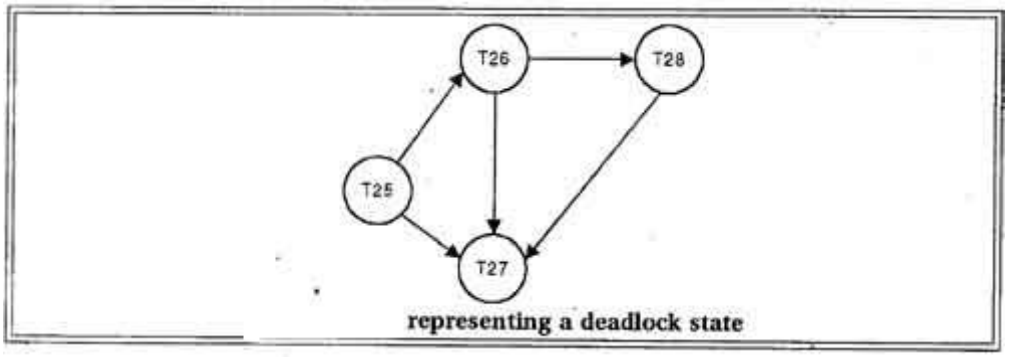
Transaction T26 is waiting for transaction T28.



representing no deadlock state

This wait-for graph has no cycle, so there is no deadlock state.

Suppose now that transaction T28 is requesting an item held by T27. Then the edge T28 ----------->T27 is added to the wait -for graph, resulting in a new system state as shown in figure.



representing a deadlock state

This time the graph contains the cycle.

T26------>T28------->T27----------->T26

It means that transactions T26, T27 and T28 are all deadlocked.

Invoking the deadlock detection algorithm

The invoking of deadlock detection algorithm depends on two factors:

• How often does a deadlock occur?

• How many transactions will be affected by the deadlock?

If deadlocks occur frequently, then the detection algorithm should be invoked more frequently than usual. Data items allocated to deadlocked transactions will be unavailable to other transactions until the deadlock can be broken. In the worst case, we would invoke the detection algorithm every time a request for allocation could not be granted immediately.

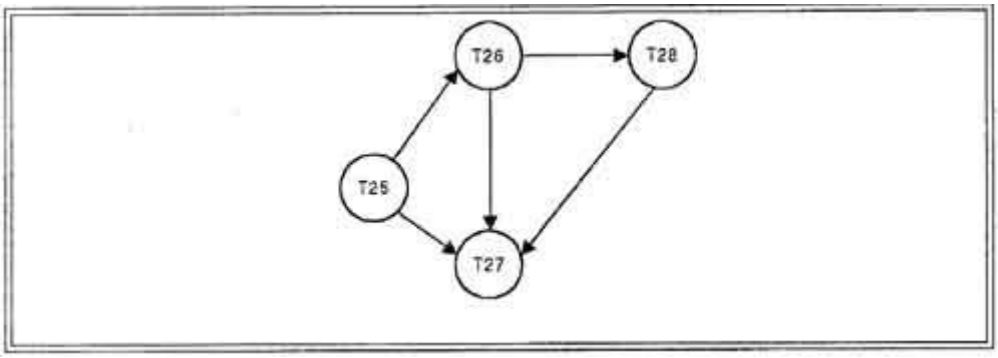
Deadlock Recovery

When a detection algorithm determines that a deadlock exists, the system must recover from the deadlock. The most common solution is to roll back one or more transactions to break the deadlock. Choosing which transaction to abort is known as Victim Selection.

Choice of Deadlock victim

In below wait-for graph transactions T26, T28 and T27 are deadlocked. In order to remove deadlock one of the transaction out of these three transactions must be roll backed.

We should roll back those transactions that will incur the minimum cost. When a deadlock is detected, the choice of which transaction to abort can be made using following criteria:



• The transaction which have the fewest locks

• The transaction that has done the least work

• The transaction that is farthest from completion

Rollback

Once we have decided that a particular transaction must be rolled back, we must determine how far this transaction should be rolled back. The simplest solution is a total rollback; Abort the transaction and then restart it. However it is more effective to roll back

the transaction only as far as necessary to break the deadlock. But this method requires the system to maintain additional information about the state of all the running system.

Problem of Starvation

In a system where the selection of victims is based primarily on cost factors, it may happen that the same transaction is always picked as a victim. As a result this transaction never completes can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

Source: https://www.youtube.com/watch?v=WZtebOyiu0M

Time Stamping Methods

Timestamp-Based Concurrency Control
In lock-based concurrency control, conflicting actions of different transactions are ordered by the order in which locks are obtained, and the lock protocol extends this ordering on actions to transactions, thereby ensuring serializability. In optimistic concurrency control, a timestamp ordering is imposed on transactions, and validation checks that all conflicting actions occurred in the same order.

Timestamps can also be used in another way: each transaction can be assigned a time stamp at startup, and we can ensure, at execution time, that if action aj of transaction Tj conflicts with action aj of transaction Tj, aj occurs before aj if TS(Ti)<TS(Tj). If an action violates this ordering, the transaction is aborted and restarted.

To implement this concurrency control scheme, every database object O, is given a read timestamp RTS(O) and a write timestamp WTS(O). If transaction T wants to read object O, and TS (T) <WTS(O), the order of this read with respect to the most recent write on O would violate the timestamp order between this transaction and the writer. Therefore, T is aborted and restarted with a new, larger timestamp. If TS(T) >WTS(O), T reads O, and RTS(O) is set to the larger of RTS(O)and TS(T). (Note that there is a physical

change—the change to RTS(O)—to be written to disk and to be recorded in the log for recovery purposes, even on reads. This write operation is a significant overhead.)

Observe that if T is restarted with the same timestamp, it is guaranteed to be aborted again, due to the same conflict. Contrast this behavior with the use of timestamps in 2PL for deadlock prevention: there, transactions were restarted with the same timestamp as before in order to avoid repeated restarts. This shows that the two uses of timestamps are quite different and should not be confused.

Next, let us consider what happens when transaction T wants to write object O:

i) If TS(T)<RTS(O), the write action conflicts with the most recent read action of O, and T is therefore aborted and restarted.
ii) If TS(T) <WTS(O), a naive approach would be to abort T because its write action conflicts with the most recent write of O and is out of timestamp order. It turns out that we can safely ignore such writes and continue. Ignoring outdated writes is called the Thomas Write Rule.
iii) Otherwise, T writes O and WTS (O) is set to TS(T).

Optimistic Methods

Optimistic concurrency control (**OCC**) is a concurrency control method applied to transactional systems such as relational database management systems and software transactional memory. OCC assumes that multiple transactions can frequently complete without interfering with each other. While running, transactions use data resources without acquiring locks on those resources. Before committing, each transaction verifies that no other transaction has modified the data it has read. If the check reveals conflicting modifications, the committing transaction rolls back and can be restarted.[1] Optimistic concurrency control was first proposed by H.T. Kung.[2]

OCC is generally used in environments with low data contention. When conflicts are rare, transactions can complete without the expense of managing locks and without having transactions wait for other transactions' locks to clear, leading to higher throughput than other concurrency control methods. However, if contention for data resources is frequent, the cost of repeatedly restarting transactions hurts performance significantly; it is commonly thought[*who?*] that other concurrency control methods have better performance under these conditions.[*citation needed*] However, locking-based ("pessimistic") methods also can deliver poor performance because locking can drastically limit effective concurrency even when deadlocks are avoided.

More specifically, OCC transactions involve these phases:

- **Begin**: Record a timestamp marking the transaction's beginning.
- **Modify**: Read database values, and tentatively write changes.
- **Validate**: Check whether other transactions have modified data that this transaction has used (read or written). This includes transactions that completed after this transaction's start time, and optionally, transactions that are still active at validation time.

- **Commit/Rollback**: If there is no conflict, make all changes take effect. If there is a conflict, resolve it, typically by aborting the transaction, although other resolution schemes are possible. Care must be taken to avoid a TOCTTOU bug, particularly if this phase and the previous one are not performed as a single atomic operation

DATABASE RECOVERY MANAGEMENT

A computer system, like any other device, is subject to failure from a variety of causes: disk crash, power outage, software error, a fire in the machine room, even sabotage. In any failure, information may be lost. Therefore, the database system must take actions in advance to ensure that the atomicity and durability properties of transactions, introduced in Chapter 15, are preserved. An integral part of a database system is a recovery scheme that can restore the database to the consistent state that existed before the failure. The recovery scheme must also provide high availability; that is, it must minimize the time for which the database is not usable after a crash.

Failure Classification

There are various types of failure that may occur in a system, each of which needs to be dealt with in a different manner. The simplest type of failure is one that does not result in the loss of information in the system. The failures that are more difficult to deal with are those that result in loss of information. In this chapter, we shall consider only the following types of failure :

- **Transaction failure.** There are two types of errors that may cause a transaction to fail :
  - **Logical error.** The transaction can no longer continue with its normal execution because of some internal condition, such as bad input, data not found, overflow, or resource limit exceeded.
  - **System error.** The system has entered an undesirable state (for example, deadlock), as a result of which a transaction cannot continue with its normal execution. The transaction, however, can be re-executed at a later time.
- **System crash.** There is a hardware malfunction, or a bug in the database software or the operating system, that causes the loss of the content of volatile storage, and brings transaction processing to a halt. The content of nonvolatile storage remains intact, and is not corrupted.
  The assumption that hardware errors and bugs in the software bring the system to a halt, but do not corrupt the nonvolatile storage contents, is known as the fail-stop assumption. Well-designed systems have numerous internal checks, at the hardware and the software level, that bring the system to a halt when there is an error. Hence, the fail-stop assumption is a reasonable one.
- **Disk failure.** A disk block loses its content as a result of either a head crash or failure during a data transfer operation. Copies of the data on other disks, or archival backups on tertiary media, such as tapes, are used to recover from the failure.

To determine how the system should recover from failures, we need to identify the failure modes of those devices used for storing data. Next, we must consider how these failure modes affect the contents of the database. We can then propose algorithms to ensure

database consistency and transaction atomicity despite failures. These algorithms, known as recovery algorithms, have two parts:

i) Actions taken during normal transaction processing to ensure that enough information exists to allow recovery from failures.
ii) Actions taken after a failure to recover the database contents to a state that ensures database consistency, transaction atomicity, and durability.

STORAGE STRUCTURE

The various data items in the database may be stored and accessed in a number of different storage media. To understand how to ensure the atomicity and durability properties of a transaction, we must gain a better understanding of these storage media and their access methods.

Storage Types

We saw that storage media can be distinguished by their relative speed, capacity, and resilience to failure, and classified as volatile storage or nonvolatile storage. We review these terms, and introduce another class of storage, called stable storage.

- **Volatile storage.** Information residing in volatile storage does not usually survive system crashes. Examples of such storage are main memory and cache memory. Access to volatile storage is extremely fast, both because of the speed of the memory access itself, and because it is possible to access any data item in volatile storage directly.
- **Nonvolatile storage.** Information residing in nonvolatile storage survives system crashes. Examples of such storage are disk and magnetic tapes. Disks are used for online storage, whereas tapes are used for archival storage. Both, however, are subject to failure (for example, head crash), which may result in loss of information. At the current state of technology, nonvolatile storage is slower than volatile storage by several orders of magnitude. This is because disk and tape devices are electromechanical, rather than based entirely on chips, as is volatile storage. In database systems, disks are used for most nonvolatile storage. Other nonvolatile media are normally used only for backup data. Flash storage, though nonvolatile, has insufficient capacity for most database systems.
- **Stable storage.** Information residing in stable storage is never lost (never should be taken with a grain of salt, since theoretically never cannot be guaranteed — for example, it is possible, although extremely unlikely, that a black hole may envelop the earth and permanently destroy all data!). Although stable storage is theoretically impossible to obtain, it can be closely approximated by techniques that make data loss extremely unlikely.

The distinctions among the various storage types are often less clear in practice than in our presentation. Certain systems provide battery backup, so that some main memory can survive system crashes and power failures. Alternative forms of nonvolatile storage, such as optical media, provide an even higher degree of reliability than do disks.

Stable-Storage Implementation

To implement stable storage, we need to replicate the needed information in several nonvolatile storage media (usually disk) with independent failure modes, and to update the information in a controlled manner to ensure that failure during data transfer does not damage the needed information.

Recall that RAID systems guarantee that the failure of a single disk (even during data transfer) will not result in loss of data. The simplest and fastest form of RAID is the mirrored disk, which keeps two copies of each block, on separate disks. Other forms of RAID offer lower costs, but at the expense of lower performance.

RAID systems, however, cannot guard against data loss due to disasters such as fires or flooding. Many systems store archival backups of tapes off-site to guard against such disasters. However, since tapes cannot be carried off-site continually, updates since the most recent time that tapes were carried off-site could be lost in such a disaster. More secure systems keep a copy of each block of stable storage at a remote site, writing it out over a computer network, in addition to storing the block on a local disk system. Since the blocks are output to a remote system as and when they are output to local storage, once an output operation is complete, the output is not lost, even in the event of a disaster such as a fire or flood.

In the remainder of this section, we discuss how storage media can be protected from failure during data transfer. Block transfer between memory and disk storage can result in

- **Successful completion.** The transferred information arrived safely at its destination.
- **Partial failure.** A failure occurred in the midst of transfer, and the destination block has incorrect information.
- **Total failure.** The failure occurred sufficiently early during the transfer that the destination block remains intact.

We can extend this procedure easily to allow the use of an arbitrarily large number of copies of each block of stable storage. Although a large number of copies reduces the probability of a failure to even lower than two copies do, it is usually reasonable to simulate stable storage with only two copies.

Data Access

The database system resides permanently on nonvolatile storage (usually disks), and is partitioned into fixed-length storage units called blocks. Blocks are the units of data transfer to and from disk, and may contain several data items. We shall assume that no data item spans two or more blocks. This assumption is realistic for most data-processing applications, such as our banking example.

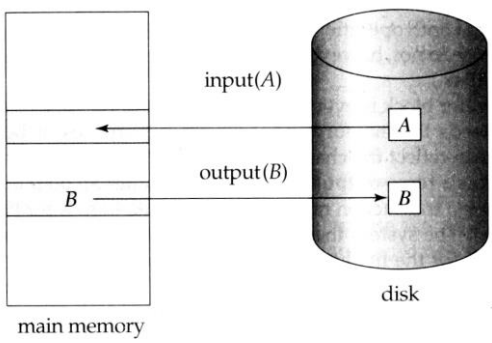


Fig. 4 : Block storage operations.

Transactions input information from the disk to main memory, and then output the information back onto the disk. The input and output operations are done in block units. The blocks residing on the disk are referred to as physical blocks; the blocks residing temporarily in main memory are referred to as buffer blocks. The area of memory where blocks reside temporarily is called the disk buffer.

Block movements between disk and main memory are initiated through the following two operations:

1) Input (B) transfers the physical block B to main memory.
2) Output (B) transfers the buffer block B to the disk, and replaces the appropriate physical block there.

Each transaction T_i has a private work area in which copies of all the data items accessed and updated by T_i are kept. The system creates this work area when the transaction is initiated; the system removes it when the transaction either commits or aborts. Each data item X kept in the work area of transaction T_i is denoted by x_i. Transaction T_i interacts with the database system by transferring data to and from its work area to the system buffer. We transfer data by these two operations:

i) read (X) assigns the value of data item X to the local variable x_i. It executes this operation as follows:
   (a) If block B_X on which X resides is not in main memory, it issues input(B_X).
   (b) It assigns to x_i, the value of X from the buffer block.

ii) write(X) assigns the value of local variable x_i to data item X in the buffer block. It executes this operation as follows:
   (a) If block B_X on which X resides is not in main memory, it issues input (B_X).
   (b) It assigns the value of x_i to X in buffer B_X.

Note that both operations may require the transfer of a block from disk to main memory. They do not, however, specifically require the transfer of a block from main memory to disk.

A buffer block is eventually written out to the disk either because the buffer manager needs the memory space for other purposes or because the database system wishes to reflect the change to B on the disk. We shall say that the database system performs a force-output of buffer B if it issues an output (B).

When a transaction needs to access a data item X for the first time, it must execute read (X). The system then performs all updates to X on x_i. After the transaction accesses X for the final time, it must execute write(X) to reflect the change to X in the database itself.

The output (B_X) operation for the buffer block B_X on which X resides does not need to take effect immediately after write (X) is executed, since the block B_X may contain other data items that are still being accessed. Thus, the actual output may take place later. Notice that, if the system crashes after the write(X) operation was executed but before output(B_X) was executed, the new value of X is never written to disk and, thus, is lost.

Recovery and Atomicity

Consider again our simplified banking system and transaction T_i that transfers \$50 from account A to account B, with initial values of A and B being \$1000 and \$2000,

respectively. Suppose that a system crash has occurred during the execution of T_i, after output(B_A) has taken place, but before output(B_B) was executed, where B_A and denote the buffer blocks on which A and B reside. Since the memory contents were lost, we do not know the fate of the transaction; thus, we could invoke one of two possible recovery procedures:

- Re-execute T_i. This procedure will result in the value of A becoming $900, rather than $950. Thus, the system enters an inconsistent state.
- Do not re-execute T_i. The current system state has values of $950 and $2000 for A and B, respectively. Thus, the system enters an inconsistent state.

In either case, the database is left in an inconsistent state, and thus this simple recovery scheme does not work. The reason for this difficulty is that we have modified the database without having assurance that the transaction will indeed commit. Our goal is to perform either all or no database modifications made by T_i. However, if T_i performed multiple database modifications, several output operations may be required, and a failure may occur after some of these modifications have been made, but before all of them are made.

To achieve our goal of atomicity, we must first output information describing the modifications to stable storage, without modifying the database itself. As we shall see, this procedure will allow us to output all the modifications made by a committed transaction, despite failures.

Questions:

1.Explain the concept of transaction
2.Describe ACID properties of transaction
3.Explain difference between the terms serial serial schedule and serializable schedule with suitable examples.
4.Explain View and conflict serializability with suitable example
5.Explain the need of concurrency control in transaction management
6.Write a short note on Two phase locking protocol
7.Explain Timestamp based protocol
8.Show that two phase locking protocol ensures conflict serializability
9.What is Deadlock.Explain Deadlock detection
10.Explain Deadlock Recovery

Multiple Choice Questions

1.Identify the characteristics of transactions
a)Atomicity
b)Durability
c)Isolation
d) All of the mentioned

2.Which of the following has “all-or-none” property ?
a)Atomicity

b)Durability
c)Isolation
d) All of the mentioned

3.The database system must take special actions to ensure that transactions operate properly without interference from concurrently executing database statements. This property is referred to as
a)Atomicity
b)Durability
c)Isolation
d) All of the mentioned

4. Deadlocks are possible only when one of the transactions wants to obtain a(n) ____ lock on a data item.
a)binary
b)exclusive
c)shared
d)complete

5. The ____ statement is used to end a successful transaction.
a)COMMIT
b)DONE
c)END
d)QUIT

6. If a transaction acquires a shared lock, then it can perform ..... operation.
a) read
b) write
c) read and write
d) update

7. A system is in a _____ state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set.
a)Idle
b)Waiting
c)Deadlock
d)Ready

8. The deadlock state can be changed back to stable state by using _____ statement.
a)Commit
b)Rollback
c)Savepoint
d)Deadlock

9.What are the ways of dealing with deadlock ?
a)Deadlock prevention

b)Deadlock recovery
c)Deadlock detection
d)All of the mentioned

10. The situation where the lock waits only for a specified amount of time for another lock to be released is
a) Lock timeout
b) Wait-wound
c) Timeout
d) Wait