T.E. Computer

Database Management Systems

Unit-V

By

Prof. Dr. K.P. Adhiya

SSBT's COET, Bambhori-Jalgaon

Acknowledgment

1. Abraham Silberschatz, Henry F. Korth, S. Sudarshan, “Database System Concepts”, 6th Edition, McGraw-Hill Hill Education.
2. Abraham Silberschatz, Henry F. Korth, S. Sudarshan, “Database System Concepts”, 4th Edition, McGraw-Hill Hill Education.
3. Ramez Elmasri and Shamkant B. Navathe “Fundamentals of Database Systems”, 5th Edition, Pearson.
4. Express Learning, “Database Management Systems”, ITL Education Solutions Limited.
5. Archana Verma, “Database Management Systems”, GenNext Publication.
6. Dr. Rajiv Chopra, “Database Management Systems (DBMS) – A Practical Approach”, 5th Edition, S. Chand Technical
7. Tanmay Kasbe, “Database Management System Concepts – A Practical Approach”, First Edition, Educreation Publishing.
8. Mahesh Mali, “Database Management Systems”, Edition 2019, TechKnowledge Publications.
9. Rajendra Prasad Mahapatra, Govind Verma, “Database Management System”, Khanna Publishing.
10. Malay K. Pakhira, “Database Management System”, Eastern Economy Edition, PHI.
11. Sarika Gupta, Gaurav Gupta, “Database Management System”, Khanna Book Publishing Edition.
12. Riktesh Srivastava, Rajita Srivastava, “Relational Database Management System”, New Age International Publishers.
13. Peter Rob, Carlos Coronel, “Database System Concepts’, Cenage Learning, India Edition
14. Bipin C. Desai, “An Introduction to Database Systems”, Galgotia Publications.
15. G.K. Gupta, “Database Management Systems”, McGraw Hill Education.
16. Shio Kumar Singh, “Database Systems – Concepts, Design and Applications”, 2nd Edition, PEARSON.
17. S.D.Joshi, “Database Management System”, Tech-Max Publication.
18. R. Ramkrishnan , J. Gehrke, "Database Management Systems", 3rd Edition, McGraw-Hill
19. C. J. Date, “Introduction to Database Management Systems”, 8th Edition, Pearson
20. Atul Kahate, “Introduction to Database Management System”, 3rd Edition, Pearson.
21. Bharat Lohiya, “Database Systems”, Tenth Edition, Aditya Publication, Amravati.
22. Vijay Krishna Pallaw, “Database Management System”, 2nd, Asian Books Pvt. Ltd.
23. Database Management Systems, Database Management Systems.
24. Mrs. Jyoti G. Mante (Khurpade), Mrs. Smita M. Dandge, “Database Mangement System”, Nirali Prakashan.
25. Step by Step Database Systems (DBMS), Shiv Krupa Publications, Akola

Prof. Dr. K.P. Adhiya

26. Mrs. Sheetal Gujar –Takale, Mr. Sahil K. Shah, “Database Management System”, Nirali Prakashan.
27. Mrs. Jyoti G. Mante (Khurpade), U.S. Shirshetti, M.V. Salvi, K.S. Sakure, “Relational Database Management System”, Nirali Prakashan.
28. Seema Kedar, Rakesh Shirsath, “Database Management Systems”, Technical Publications.
29. Pankaj B. Brahmankar, “Database Management Systems”, Tech-Max Publications, Pune.
30. Imran Saeed, Tasleem Mustafa, Tariq Mahmood, Ahsan Raza Sattar, “A Fundamental Study of Database Management Systems”, 3rd Edition, IT Series Publication.
31. Database Management Systems Lecture Notes, Malla Reddy College of Engineering and Technology, Secunderabad.
32. Dr. Satinder Bal Gupta, Aditya Mittal, “Introduction to Database Management System, University Science Press.
33. E-Notes BCS 41/ BCA 41 on “Database Management System”, Thiruvalluvar University.
34. Bighnaraj Naik, Digital Notes on “Relational Database Management System”, VSSUT, Burla.
35. Viren Sir, Relational database Management System”, Adarsh Institute of Technolgoyt (Poly), VITA.
36. Sitansu S. Mitra, “Principles of Relational Database Systems”, Prentice Hall.
37. Neeraj Sharma, Liviu Perniu, Raul F. Chong, Abhishek Iyer, Chaitali Nandan, Adi-Cristina Mitea, Mallarswami Nonvinkere, Mirela Danubianu, “Database Fundamentals”, First Edition, DB2 On Campus Book Series.
38. Database Management System, Vidyavahini First Grade College, Tumkur.
39. Bhavna Sangamnerkar, Revised by: Shiv Kishor Sharma, “Database Management System”, Think Tanks Biyani Group of Colleges.
40. Tibor Radvanyi, “Database Management Systems”.
41. Ramon A. Mata-Toledo, Pauline K. Cushman, “Fundamentals of Relational Databases”, Schaum’s Outlies.
42. P.S. Gill, “Database Management Systems”, 2nd Edition, Dreamtech Press, WILEY

Acknowledgment

Web Resources:-

https://www.exploredatabase.com/2016/04/database-transaction-states-in-dbms.html
https://www.javatpoint.com/dbms-log-based-recovery
https://www.geeksforgeeks.org/difference-between-deferred-update-and-immediate-update/

Unit-V

(Transactions)

➢ Transaction Concept

- Often, a collection of several operations on the database appears to be a single unit from the point of view of the database user. Within the database system, however, it consists of several operations.
- The collection of operations that form a single logical unit of work is called a **transaction.**
- A database system must ensure proper execution of transactions despite failures—**either the entire transaction executes, or none of it does.**
- The database system must manage concurrent execution of transactions in a way that avoids the introduction of inconsistency.
- A transaction is delimited by statements (or function calls) of the form **begin transaction** and **end transaction**. The transaction consists of all operations executed between the **begin transaction** and **end transaction**.
- The collection of steps must appear to the user as a single, **indivisible unit**. Since a transaction is indivisible, it **either executes in its entirety or not at all.**
- This "all-or-none" property is referred to as **atomicity.**
- 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 **isolation**.
- Even if the system ensures correct execution of a transaction, this serves little purpose if the system subsequently crashes and, as a result, the system "forgets" about the transaction. Thus, a transaction's actions must persist across crashes. This property is referred to as **durability**.
- The database must preserve database consistency.

- To ensure the integrity of the data, the database system must maintain some desirable properties of the transaction (i.e. Every transaction must have some characteristics). These properties are known as **ACID** properties, the short form derived from the first letter of the term **Atomicity, Consistency, Isolation, and Durability**.
    - **Atomicity**: - A transaction should be treated as a single unit of operation. Atomicity implies that either all of the operations that make up a transaction should execute or none of them should occur.
    - **Consistency**: - It implies that if all the operations of a transaction are executed completely, the database is transformed from one consistent state to another consistent state.
    - **Isolation: -** Even though multiple transactions may execute concurrently, isolation implies that each transaction appears to run in isolation with other. Thus each transaction is unaware of other transaction and do not interfere with each other.
    - **Durability: -** It is also known as Permanence. It implies that once a transaction is completed successfully, the changes made by the transaction persist (i.e. remains permanent) in the database, even if the system fails.

➢ A Simple Transaction Model

- For example, consider a transaction T_i that transfers $100 from account A to account B.
- Transactions access data using two operations:

  read(A) and write(A)
- Let–

  Initial values of account A is $2000

  Initial values of account B is $1500
- The sum of the values of account A and B is $3500 before the transaction. The sum of the values of account A and B should be $3500 after the execution of transaction.

- This transaction T_i can be written as follows:

```
T_i: read(A);
        A := A – 100;
        write(A);
        read(B);
        B := B + 100;
        write(B);
```

- **Atomicity Requirement**: - If transaction T_i is executed, either \$100 should be transferred from account A to account B or neither of the accounts should be affected. If transaction T_i fails after debiting \$100 from account A, but before crediting \$100 to account B. So sum of values of account A and B will be \$1900+\$1500=\$3400, as the system destroys \$100 as a result of the failure. It means the sum A+B is not preserved.

  Thus, because of the failure, the state of the system no longer reflects a real state of the world that the database is supposed to capture. Such a state is an **inconsistent state**. Such inconsistencies should not be present in a database system.

  That is the reason for the **atomicity requirement:** If the atomicity property is present, all actions of the transaction are reflected in the database, or none are.

  Ensuring atomicity is the responsibility of the database system; specifically, it is handled by a component of the database called the **recovery system.**

- **Consistency Requirement: -** The consistency requirement here is that the sum of *A* and *B* be unchanged by the execution of the transaction. Without the consistency requirement, money could be created or destroyed by the transaction.

  Here in above example, sum of the values of account A and B is \$3500 before the transaction and it should be \$3500 after the execution of transaction.

  So the **consistency requirement:** if the database is consistent before an execution of the transaction, the database remains consistent after the execution of the transaction.

- **Durability Requirement**: - once the user has been notified that the transaction has completed (i.e., the transfer of the $100 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.
  The durability can be guaranteed by ensuring that either –
  - All the changes made by the transaction are written to the disk before the transaction is completed or
  - The information about all the changes made by the transaction is written to the disk and is sufficient to enable the database system to reconstruct the changes when the database system is restarted after the failure.
- **Isolation Requirement**: - If several transactions are executed concurrently, their operations may interleave in some undesirable way, resulting in an inconsistent state.
  For example, as we saw earlier, the database is temporarily inconsistent while the transaction to transfer funds from *A* to *B* is executing, with the deducted total written to *A* and the increased total yet to be written to *B*. If a second concurrently running transaction reads *A* and *B* at this intermediate point and computes *A*+*B*, it will observe an inconsistent value. Furthermore, if this second transaction then performs updates on *A* and *B* based on the inconsistent values that it read, the database may be left in an inconsistent state even after both transactions have completed.
  A way to avoid the problem of concurrently executing transactions is to execute transactions serially—**that is, one after the other.**

➢ Transaction Atomicity and Durability

- A transaction that completes its execution successfully is said to be **committed.**
- The **State Transition Diagram** is a diagram that describes how a transaction passes through various states during its execution.
- Whenever a transaction is submitted to DBMS for execution, either it executes successfully or fails due to some reasons. During its

execution, a transaction passes through various states that are active, partially committed, committed, failed and terminated.

- The following figure 5.1 shows the State Transition Diagram showing various states of a transaction:

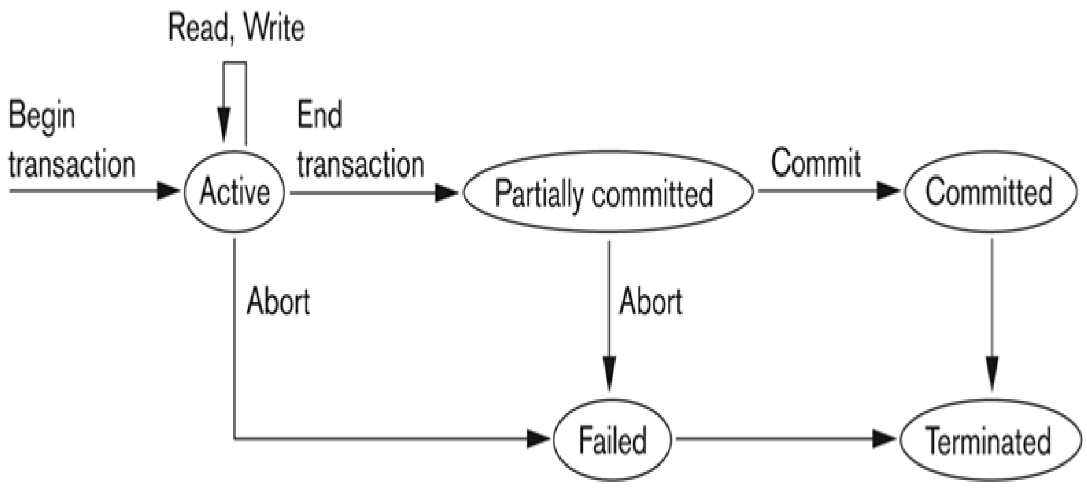


Figure 5.1:- State Transition Diagram of a transaction

-

```
BEGIN_TRANSACTION
        READ/WRITE Statements
END_TRANSACTION
COMMIT_TRANSACTION
```

- The transaction must be in one of the following states:
  - **Active: -** the initial state; the transaction stays in this state while it is executing several READ and WRITE operations.
  - **Partially committed: -** Once the transaction executes its final operation, the system marks END_TRANSACTION operation to specify the end of the transaction execution. At this point, the transaction enters into the **Partially Committed state.**
  - **Committed: -** When the transaction is complete and COMMIT_TRANSACTION statement is given, then it indicates the successful end of the transaction. Now the transaction is said to be **committed** and all its changes are reflected permanently in the database.

- **Failed**: - If the transaction is aborted during its active state or the system fails to write the changes in the log file, the transaction enters into the **failed** state.
- **Terminated: -** When the transaction leaves the system, it enters into the **terminated** state.

- Other State Transition Diagram is as shown in figure 5.2:

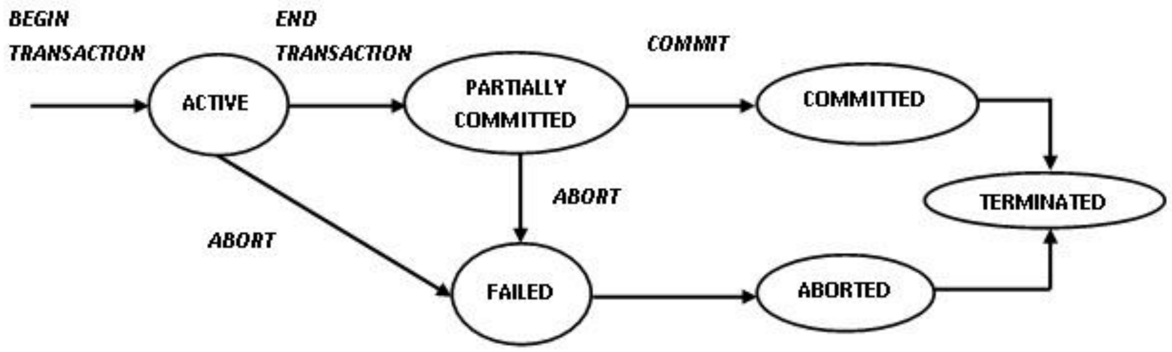


Figure 5.2:- State Transition Diagram of a transaction

Aborted: - after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction. Two options after it has been aborted:

- restart the transaction –> only if no internal logical error
- kill the transaction

Unit-V

(Concurrency Control)

➢ Lock-Based Protocols

- When several transactions execute concurrently in the database, the isolation property may no longer be preserved. To ensure that it is, the system must control the interaction among the concurrent transactions; this control is achieved through one of a variety of mechanisms called ***concurrency control schemes***.
- **Concurrency Control Protocols**
  - To ensure when to give access to data item when transactions are getting executed concurrently or in interleaved way.
- One way to ensure isolation is to require that data items be accessed in a **mutually exclusive manner**; that is, while one transaction is accessing a data item, no other transaction can modify that data item. The most common method used to implement this requirement is to allow a transaction to access a data item only if it is currently holding a **lock** on that item.

✓ Locks

- A **lock** is a variable associated with each data item that indicates whether a read or write operation can be applied to the data item.
- A lock controls the concurrent access and maintains the consistency and integrity of the database.
- i.e. Locks are used to ensure consistency.
- Database systems mainly use two modes of locking (i.e. Mainly, there are two types of locks):
  - **Shared lock** (denoted by S):- it can be acquired on a data item when a transaction wants to only read a data item and not modify it. Hence it is also known as **read lock.** If a transaction T_i has acquired a shared lock on data item Q, then T_i can read

but cannot write on Q. In addition, any number of transactions can acquire shared locks on Q simultaneously. However, no transaction can acquire an exclusive lock on Q.

- **Exclusive lock** (denoted by X):- is the commonly used locking strategy that provides an exclusive control on the data item. A transaction that wants to read as well as write a data item must acquire an exclusive lock on the data item. Hence, an exclusive lock is also known as the **write lock** or **update lock.** If a transaction T_i has acquired an exclusive lock on a data item Q, then no other transaction is allowed to access Q until T_i releases its lock on Q.

- The use of these two locks allow multiple transactions to read a data item but limits write access to just one transaction at a time.
- The compatibility matrix of shared and exclusive locks is shown in the figure 5.3:

	S	X
S	true	false
X	false	false

Figure 5.3:- Lock-compatibility matrix

- The same thing is explained given in following figure 5.4

		User 1, HOLDER of the LOCK on data item X	
		Shared or Read Lock	Write or Exclusive Lock
User2, REQUESTOR of LOCK for data item X	Shared or Read Lock	**YES**: USER 2 will granted with Read lock	**NO**: USER 2 will not be granted with Read lock
	Write or Exclusive Lock	**NO**: USER 2 will not be granted with Write lock	**NO**: USER 2 will not be granted with Write lock

Figure 5.4

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions.

- Any number of transactions can hold shared locks on an item,
    - But if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.
- A transaction requests a shared lock on data item *Q* by executing the **lock- S(*Q*)** instruction. Similarly, a transaction requests an exclusive lock through the **lock-X(*Q*)** instruction. A transaction can unlock a data item *Q* by the **unlock(*Q*)** instruction.
- Consider again the banking example. Let A and B be two accounts that are accessed by transactions T1 and T2. Transaction T1 transfers $50 from account B to account A(figure 5.5).Transaction T2 displays the total amount of money in accounts A and B—that is, the sum A + B (Figure 5.6).

```
T1:lock-X(B);
    read(B);
    B := B − 50;
    write(B);
    unlock(B);
    lock-X(A);
    read(A);
    A := A + 50;
    write(A);
    unlock(A).
```

Figure 5.5:- Transaction T_1

```
T2: lock-S(A);
    read(A);
    unlock(A);
    lock-S(B);
    read(B);
    unlock(B);
    display(A + B).
```

Figure 5.6:- Transaction T_2

- Suppose that the values of accounts *A* and *B* are \$100 and \$200, respectively. If these two transactions are executed serially, either in the order *T*1, *T*2 or the order *T*2, *T*1, then transaction *T*2 will display the value \$300. If, however, these transactions are executed concurrently, then schedule 1, in Figure 5.7, is possible. In this case, transaction *T*2 displays \$250, which is incorrect. The reason for this mistake is that the transaction *T*1 unlocked data item *B* too early, as a result of which *T*2 saw an inconsistent state.

T_1	T_2	Concurrency-control manager
lock-X(*B*)		
		grant-X(*B*, *T*1)

read(*B*)
B := *B* − 50
write(*B*)
unlock(*B*)		

	lock-S(*A*)	
		grant-S(*A*, *T*2)

	read(*A*)
	unlock(*A*)
	lock-S(*B*)	

		grant-S(*B*, *T*2)

	read(*B*)
	unlock(*B*)
	display(*A* + *B*)	

lock-X(*A*)		
		grant-X(*A*, *T*1)

read(*A*)
A := *A* + 50
write(*A*)
unlock(*A*)		

Figure 5.7: Schedule 1.

- Suppose now that unlocking is delayed to the end of the transaction. Transaction T_3 corresponds to T_1 with unlocking delayed (Figure 5.8). Transaction T_4 corresponds to T_2 with unlocking delayed (Figure 5.9).
- The sequence of reads and writes in schedule 1 (figure 5.7), which lead to an incorrect total of $250 being displayed, is no longer possible with T_3 and T_4.
- The figures 5.8 and 5.9 are as follows:

```
T3:lock-X(B);
    read(B);
    B := B − 50;
    write(B);
    lock-X(A);
    read(A);
    A := A + 50;
    write(A);
    unlock(B);
    unlock(A).
```

Figure 5.8:- Transaction T_3 (transaction T_1 with unlocking delayed).

```
T4: lock-S(A);
    read(A);
    lock-S(B);
    read(B);
    display(A + B);
    unlock(A);
    unlock(B).
```

Figure 5.9:- Transaction T_4 (transaction T_2 with unlocking delayed).

- Unfortunately, locking can lead to an undesirable situation. Consider the partial schedule of Figure 5.10 for T_3 and T_4. Since T_3 is holding an exclusive mode lock on B and T_4 is requesting a shared-mode lock on B, T_4 is waiting for T_3 to unlock B. Similarly, since T_4 is holding a shared-mode lock on A and T_3 is requesting an exclusive-mode lock on A, T_3 is waiting for T_4 to unlock A. Thus, we have arrived at a state where neither of these transactions can ever proceed with its normal execution. This situation is called **deadlock**. When deadlock occurs, the system must roll back one of the two transactions. Deadlocks are definitely preferable to inconsistent state, since they can be handled by rolling back transactions, whereas inconsistent states may lead to real-world problems that cannot be handled by the database system.

- The figure 5.10 is as follows:

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

Figure 5.10:- Schedule 2

- Each transaction in the system follows a set of rules, called a **locking protocol**, indicating when a transaction may lock and unlock each of the data items. Locking protocols restrict the number of possible schedules.

✓ <u>Granting of Locks</u>

- When a transaction requests a lock on a data item in a particular mode, and no other transaction has a lock on the same data item in a conflicting mode, the **lock can be granted.**
- However, care must be taken to avoid the following scenario. Suppose a transaction T_2 has a shared-mode lock on a data item, and another transaction T_1 requests an exclusive-mode lock on the data item. Clearly, T_1 has to wait for T_2 to release the shared-mode lock. Meanwhile, a transaction T_3 may request a shared-mode lock on the same data item. The lock request is compatible with the lock granted to T_2, so T_3 may be granted the shared-mode lock. At this point T_2 may release the lock, but still T_1 has to wait for T_3 to

finish. But again, there may be a new transaction T_4 that requests a shared-mode lock on the same data item, and is granted the lock before T_3 releases it. In fact, it is possible that there is a sequence of transactions that each requests a shared-mode lock on the data item, and each transaction releases the lock a short while after it is granted, but T_1 never gets he exclusive-mode lock on the data item. The transaction T_1 may never make progress, and is said to be **starved.**

- The **starvation** can be avoided by granting locks in the following manner:
  When a transaction Ti requests a lock on a data item Q in a particular mode M, the concurrency-control manager grants the lock provided that:
    - There is no other transaction holding a lock on Q in a mode that conflicts with M.
    - There is no other transaction that is waiting for a lock on Q and that made its lock request before $T_{i.}$

  Thus, a lock request will never get blocked by a lock request that is made later.

✓ The Two-Phase Locking Protocol

- One protocol that ensures serializability is the **two-phase locking protocol.**
- This protocol requires that each transaction issue lock and unlock requests in two phases:
  1. **Growing phase**. A transaction may obtain locks, but may not release any lock. It is also called as **expanding phase**. In this phase, the number of locks held by a transaction increases from zero to maximum.
  2. **Shrinking phase**. A transaction may release locks, but may not obtain any new locks. It is also called as **shrinking** or **contacting phase.** In this phase, the number of locks held by a transaction decreases from maximum to zero.

- Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase.
- For example, transactions *T*3 and *T*4 (figures 5.8 and 5.9) are two phase. While transactions *T*1 and *T*2 (figures 5.5 and 5.6) are not two phase. Note that the unlock instructions do not need to appear at the end of the transaction.
- The point in the schedule where the transaction has obtained its final lock (the end of its growing phase) is called the **lock point** of the transaction.

  (**Schedule: -** The execution sequences are called schedules. They represent the chronological order in which instructions are executed in the system. E.g. figures 5.7 and 5.10)
- Two-phase locking does not ensure freedom from deadlock. Observe that transactions T_3 and T_4 are two phase, but, in schedule 2 (Figure 5.10), they are deadlocked.
- To avoid cascading rollback in **2PL (two-phase locking),** a modification of 2PL called **strict two-phase locking** can be used. In this, a transaction does not release any of its exclusive locks until it commits or aborts.
- The **Rigorous two-phase locking** is another more restrictive version of 2PL. The benefit of this version is that a transaction does not release any of its locks (both exclusive and shared) until it commits or aborts in it.
- Strict two-phase locking and Rigorous two-phase locking (with lock conversions) are used extensively in commercial database systems.

➢ Timestamp-Based Protocols

- The two-phase locking protocol guarantees conflict serializability schedules the only problem is that if it locks an item, that item cannot be used by another transaction until it is unlocked.

- The timestamp based protocol follows a different way of solving the problem, ensuring conflict serializability **without locking**. Here they use a timestamp and a timestamp ordering protocol lays down its rules.

✓ Timestamps

- With each transaction *Ti* in the system, a unique fixed **timestamp** is associated. This timestamp is assigned by the database system before the transaction *Ti* starts execution. There are two simple methods for implementing this timestamp scheme:
  1. Use the value of System Clock (current time).
  2. Use a logical counter that is incremented after a new timestamp has been assigned.

  The timestamp value is used just to indicate the order in which the transaction arrived **early** and which was **late** in relation to each other.
- The timestamp ordering protocol does not use any locks, so there is no question that a deadlock will occur and hence the protocol is free from deadlock situation.
- To implement the timestamp ordering protocol, following are the type of timestamps that are going to be used (here W-timestamp(Q) and R-timestamp(Q) ate the two timestamp values associated with each data item Q):-
  - TS(T_i) – A unique fixed timestamp associated with the transaction T_i indicating the time before T_i begins execution.
  - **R-timestamp(Q)** – Gives the time of **latest read** of the item Q done by any transaction. E.g. if there are 2 transactions T_i and T_j and the order of operations as follows:

T_i	T_j
	Read(Q)
Read(Q)	

Then the latest read of item Q was done by transaction T_i in that case.

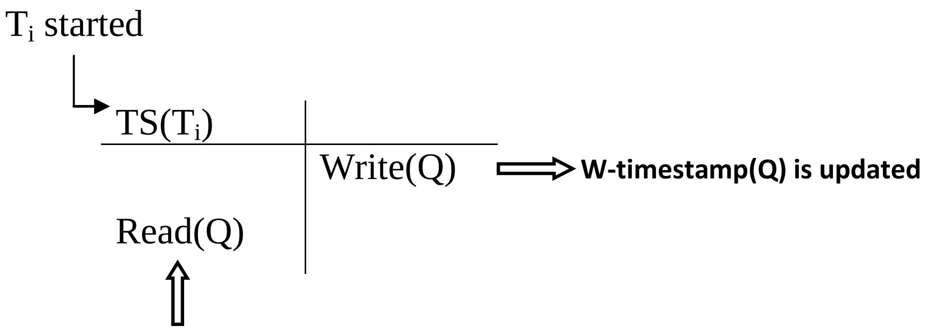
- **W-timestamp(Q)** - Gives the time of **latest write** of the item Q done by any transaction. E.g. if there are 2 transactions T_i and T_j and the order of operations as follows:

T_i	T_j
Write(Q)	
	Write(Q)

Then the latest write of item Q was done by transaction T_j in that case.

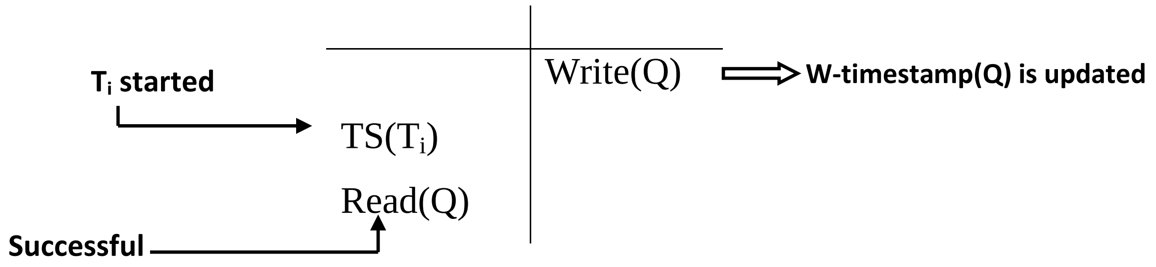
✓ The Timestamp-Ordering Protocol

- The **timestamp-ordering protocol** ensures that any conflicting read and write operations are executed in timestamp order.
- Following are the ways in which the protocol operates:
  1. Let a transaction T_i issue a Read(Q) operation.
     a) If $TS(T_i) <$ W-timestamp(Q)
        That means the start of transaction T_i has happened before the latest write operation on data item(Q). Now T_i wants to read a data item (Q), but another transaction in between performs a Write(Q) operation before T_i got a chance to read it. So the transaction is rejected and T_i is rolled back.



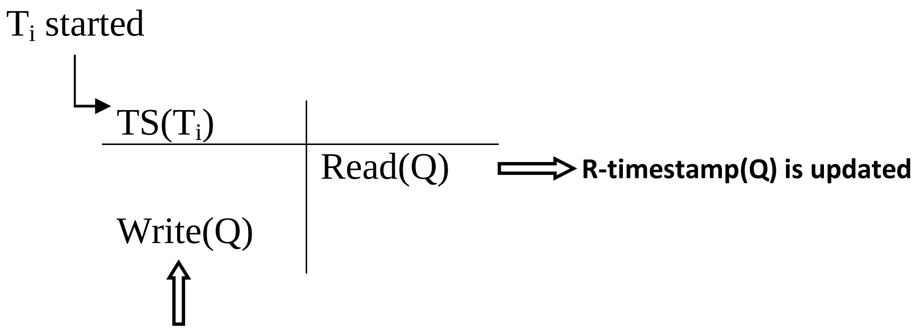
T_i started to Read(Q) but a Write(Q) happened before it and therefore T_i is not reading the correct value that it was supposed to when it started, so reject and rollback.

b) If $TS(T_i) >= W\text{-timestamp}(Q)$
That is if the transaction T_i started after the latest write operation on data item(Q) has taken place, and then you want to issue a Read(Q) operation. So there is no problem. The Read(Q) operation is reading the correct updated value, so let the transaction take place.

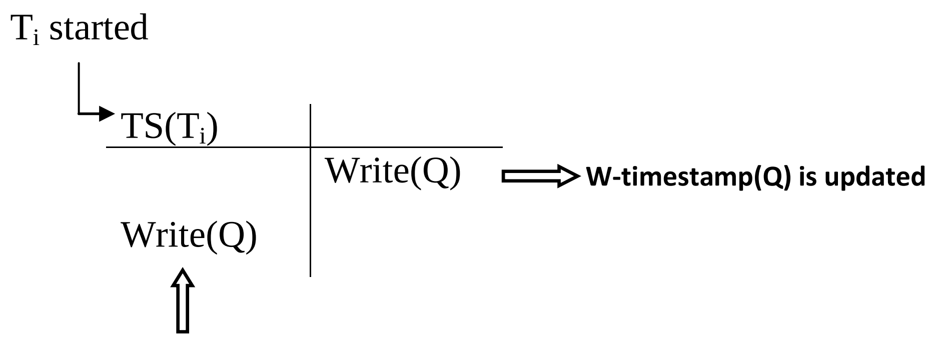


2. In the second case, now suppose a transaction wants to issue a Write(Q) operation.

a) If $TS(T_i) < R\text{-timestamp}(Q)$
Now the transaction T_i has started, and a read operation on data item (Q) takes place by another transaction before T_i gets a chance to perform a write operation on data item(Q), for which it had started. So again another transaction has read a value of Q before T_i got a chance to write it, so again reject and rollback.



b) If $TS(T_i) < W\text{-timestamp}(Q)$
   Now the transaction T_i has started before the latest write operation on data item (Q) has taken place. The T_i started to write on the data item(Q) but another transaction has written on it in between before T_i got a chance to write it. So again reject and rollback.



Unit-V

(Recovery System)

- A database is very important for any organization. It is very important to recover it as soon as possible.
- 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 failure.
- The component of DBMS that is responsible for performing the recovery operation is called **recovery manager**. The main aim of recovery is to restore the database to the most recent consistent state.

➢ Failure Classification

- There are various types of failure that may occur in a system:
    - **Transaction failure:-** There are two types of errors that may cause a transaction to fail:
        - **Logical error: -** Transaction may fail due to logical error in the transaction such as incorrect input, overflow, divide by zero, data not found, resource limit exceeded, etc.
        - **System error:-** Any undesirable state of the system such as deadlock, may also stop the normal execution of the transaction. The transaction can be re-executed at a later time.
    - **System crash (computer failure):-** Any type of hardware malfunction (e.g. RAM failure), or a bug in the database software/operating system, brings transaction processing to a halt. The content of nonvolatile storage (e.g. disk) 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**.

  - **Disk failure (Media 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 DVD or tapes, are used to recover from the failure.
  - **Physical problems and environment disasters:-**
    - Physical problems – such as theft, fire, accidental overwriting of secondary storage media etc.
    - Environmental disasters – such as floods, earthquakes etc.

- To determine how the system should recover from failures, we need to identify the failure modes of those devices used for storing data.
- **Recovery Algorithms**:-. These algorithms ensure database consistency and transaction atomicity, despite the failures. These algorithms have two parts:
  1. Action taken during normal transaction processing to ensure that enough information exists to allow recovery from failures
  2. Action taken after a failure to recover the database contents to a state that ensures database consistency, transaction atomicity, and durability

➢ Storage

- The storage media can be distinguished by their relative speed, capacity, and resilience to failure. Mainly three categories of storage:
  - Volatile storage
  - Nonvolatile storage
  - Stable storage

✓ 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.
- The **RAID** (Redundant Array of Independent Disk OR Redundant Array of Inexpensive Disk) systems guarantee that the failure of a single disk 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.
- RAID systems, 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.
- More secure systems keep a copy of each block of stable storage at a remote site,
- 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.
- If a **data-transfer failure** occurs, then the system should detect it and invoke a recovery procedure to restore the block to a consistent state. To do so, the system must maintain two physical blocks for each logical database block:
    - in the case of mirrored disks, both blocks are at the same location,
    - in the case of remote backup, one of the blocks is local, whereas the other is at a remote site.

An output operation is executed as follows:

1. Write the information onto the first physical block.

2. When the first write completes successfully, write the same information onto the second physical block.
3. The output is completed only after the second write completes successfully.

✓ Data Access

- The database system resides permanently on nonvolatile storage (usually disks) and only parts of the database are in memory at any time. The database 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.
- 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(A) transfers the physical block A to main memory.
  2. output(B) transfers the buffer block B from m.m. to the disk
- Following figure 5.11 illustrates this scheme:

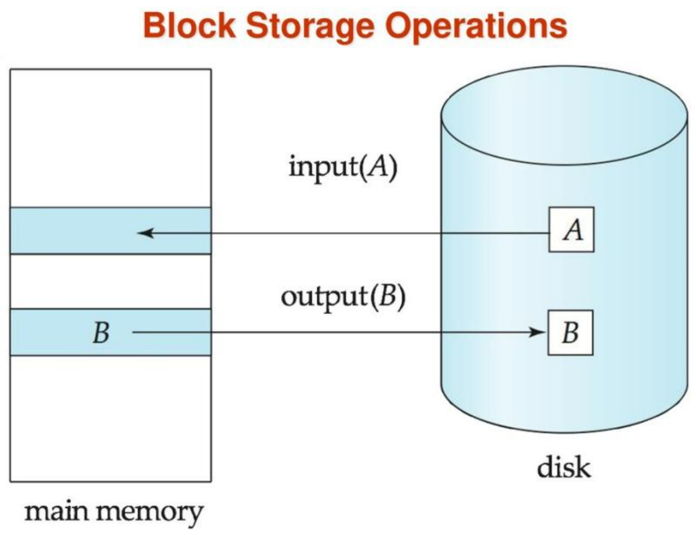


Figure 5.11:- Block Storage Operations

- A buffer block is written out to the disk:
  - either because the buffer manager needs the memory space for other purposes or
  - because the database system wants to reflect the change on the disk.

    We shall say that the database system performs a **force-output** of buffer *B* if it issues an output(*B*).

➢ Recovery and Atomicity

- Consider the banking system and a 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 B_B 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.
- When the system restarts, the value of *A* would be \$950, while that of *B* would be \$2000, which is clearly inconsistent with the atomicity requirement for transaction T_i. Unfortunately, there is no way to find out by examining the database state what blocks had been output, and what had not, before the crash.
- The goal is to perform either all or no database modifications made by T_i.

✓ Log Records

- The most widely used structure for recording database modifications is the **log**.
- The log is a sequence of **log records**, recording all the update activities in the database.
- The log is a sequence of records. Log of each transaction is maintained in some stable storage so that if any failure occurs, then it can be recovered from there.

- If any operation is performed on the database, then it will be recorded in the log.
- There are several types of log records. An **update log record** describes a single database write. It has these fields:
    - **Transaction identifier**, which is the unique identifier of the transaction that performed the write operation.
    - **Data-item identifier**, which is the unique identifier of the data item written. Typically, it is the location on disk of the data item.
    - **Old value**, which is the value of the data item prior to the write.
    - **New value,** which is the value that the data item will have after the write
- The update log record is represented as **<T_i , X_j, V_1, V_2>,**
  where T_i – transaction that has performed write operation on data item X_j.
  V_1 – value before write
  V_2 – value after write
- Other special log records are as follows:
  <T_i start>. Transaction T_i has started.
  < T_i commit>. Transaction T_i has committed.
  < T_i abort>. Transaction T_i has aborted
- Whenever a transaction performs a write, it is essential that the log record for that write be added to the log.
- Once a log record exists, we can output the modification to the database if that is desirable.
- Also, we have the ability to *undo* a modification that has already been output to the database. We undo it by using the old-value field in log records.
- E.g. Following is one of possible scenario -
  Assume that there is a transaction to modify the City of a student. The following logs are written for this transaction.
  When the transaction is initiated, then it writes 'start' log.
  <Tn, Start>

When the transaction modifies the City from 'Dhule' to 'Jalgaon', then another log is written to the file.

<Tn, City, 'Dhule', 'Jalgaon' >

When the transaction is finished, then it writes another log to indicate the end of the transaction.

<Tn, Commit>

✓ Database Modification

- A transaction creates a log record prior to modifying the database. The log records allow the system to undo changes made by a transaction, in the case of system failure, for example.
- Following are the steps taken by a transaction when modifying a data item:-
  1. The transaction performs some computations in main memory.
  2. The transaction modifies that particular data block in main memory.
  3. The database system executes the **output** operation that writes the updated data block to the disk.
- A transaction modifies the database means - it performs an update on the disk. Updates in main memory do not count as database modifications.
- The **log-based recovery** techniques maintain transaction logs to keep track of all update operations of the transactions. The log-based recovery techniques are classified into two types, techniques based on **deferred update** and techniques based on **immediate update** :
  - **Deferred-modification technique**: - It records all update operations of the transactions in the log, but postpones the execution of all the update operations until the transaction enter into the committed state (in some literature it is partial commit). It is also called NO-UNDO/REDO technique. Whenever any transaction is executed, the updates are not made immediately to the database. They are first recorded on the log file and then

those changes are applied once commit is done. This is called "Re-doing" process. Once the rollback is done none of the changes are applied to the database and the changes in the log file are also discarded. If commit is done before crashing of the system, then after restarting of the system the changes that have been recorded in the log file are thus applied to the database.

- **Immediate-modification technique: -** In this technique, as soon as the data item is modified in the main memory, the disk copy is updated. That is, the transaction is allowed to update the database in its active state.

  It is also called UNDO/REDO technique. Whenever any transaction is executed, the updates are made directly to the database and the log file is also maintained which contains both old and new values. Once commit is done, all the changes get stored permanently into the database and records in log file are thus discarded. Once rollback is done the old values get restored in the database and all the changes made to the database are also discarded. This is called **"Un-doing"** process. If commit is done before crashing of the system, then after restarting of the system the changes are stored permanently in the database.

- The transaction creates a log record prior to modifying the database. So the system has available both the **old value** prior to the modification of the data item and the **new value** that is to be written for the data item. This allows the system to perform ***undo*** and ***redo*** operations as appropriate.
  - **Undo** sets the data item specified in the log record to the old value.
  - **Redo** sets the data item specified in the log record to the new value.

✓ Concurrency Control and Recovery

- The recovery algorithms usually require that if a data item has been modified by a transaction, no other transaction can modify that data item until the first transaction commits or aborts.

- This requirement can be ensured by acquiring an exclusive lock on any updated data item and holding the lock until the transaction commits; in other words, by using strict two-phase locking.

✓ Transaction Commit

- The commit log record is the last log record of a transaction. The transaction has **committed** means this commit log record has been output to the stable storage (i.e. stored in the stable storage).
- At that point all earlier log records have already been output to stable storage. Thus, there is enough information in the log to ensure that even if there is a system crash, the updates of the transaction can be redone.
- If a system crash occurs before a log record < T_i commit> is output to stable storage, transaction T_i will be rolled back. Thus, the output of the block containing the commit log record is the single atomic action that results in a transaction getting committed.
- With most log-based recovery techniques, blocks containing the data items modified by a transaction do not have to be output to stable storage when the transaction commits, but can be output afterwards.

✓ Using the Log to Redo and Undo Transactions

- How the log can be used to recover from a system crash, and to roll back transactions during normal operation?
- For this consider – the banking example. Let T_0 be a transaction that transfers \$50 from account A to account B.
  Let initial amount in account A is \$1000, and
  Initial amount in account B is \$2000
- Transaction T_0 is as follows:-

```
T0: read(A);
    A := A − 50;
    write(A);
    read(B);
    B := B + 50;
    write(B).
```

- Let T_1 be a transaction that withdraws \$100 from account C:

  T_1: read(C);

  $C := C - 100$;

  write(C).

- The portion of the log containing the relevant information concerning these two transactions appears in figure 5.12:

  <T_0 start>

  <T_0 , A, 1000, 950>

  <T_0 , B, 2000, 2050>

  <T_0 commit>

  <T_1 start>

  <T_1 , C, 700, 600>

  <T_1 commit>

  Figure 5.12:- Portion of the system log corresponding to T_0 and T_1.

- Figure 5.13 shows one possible order in which the actual outputs took place in both the database system and the log as a result of the execution of T_0 and T_1.

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

Figure 5.13 State of system log and database corresponding to T_0 and T_1.

- Using the log, the system can handle any failure that does not result in the loss of information in nonvolatile storage. The recovery scheme uses two recovery procedures. Both these procedures make use of the log to find the set of data items updated by each transaction T_i *and* their respective old and new values.

- **redo(T_i)** sets the value of all data items updated by transaction *Ti* to the new values.
- **undo(T_i)** restores the value of all data items updated by transaction *Ti* to the old values.

Unit-V

(Database-System Architectures)

- The **architecture of a database system** is influenced by the underlying computer system on which it runs – e.g. considering some aspects such as networking, parallelism, and distribution.

➢ Centralized & Client-Server Architecture

- Centralized database systems can be those that run on a single computer system and do not interact with other computer systems.
- Client-server systems have functionality split between a server system and multiple client systems.

✓ Centralized Systems

- Centralized database systems run on a single computer system and do not interact with other computer systems.
- Such database systems may span a range from single-user database systems running on personal computers to high-performance database systems running on high-end server systems.
- A modern, general-purpose computer system consists of one to a few processors and a number of device controllers that are connected through a common bus that provides access to shared memory (**figure 5.14).**
- The processors have local cache memories that store local copies of parts of the memory, to speed up access to data.
- Each processor may have several independent **cores**, each of which can execute a separate instruction stream.
- The use of cache memory increases the overall performance of the system.

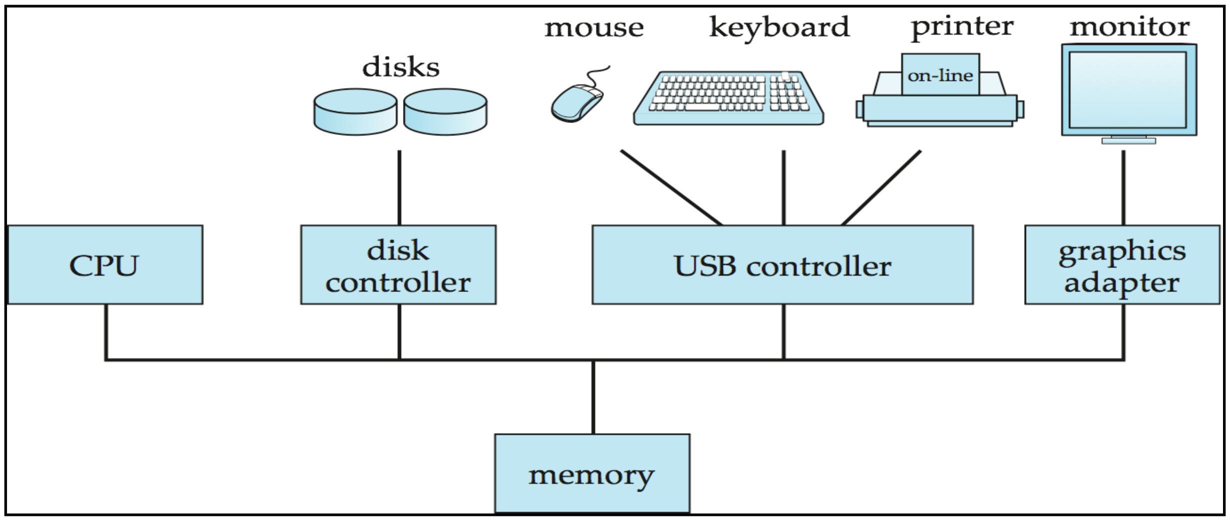


Figure 5.14:- A Centralized Computer System

- We distinguish two ways in which computers are used:
    - Single-user systems: - Personal computers and workstations fall into the first category. A typical **single-user system** is a desktop unit used by a single person, usually with only one processor and one or two hard disks, and usually only one person using the machine at a time.
    - Multiuser systems: - A typical **multiuser system** has more disks and more memory and may have multiple processors. It serves a large number of users who are connected to the system remotely.
- Database systems designed for use by single users usually do not provide many of the facilities that a multiuser database provides. In particular, they may not support concurrency control, which is not required when only a single user can generate updates. Provisions for crash recovery in such systems are either absent or primitive—for example, they may consist of simply making a backup of the database before any update. The database systems designed for multiuser systems support the full transactional features.
- Most general-purpose computer systems in use today have multiple processors, they have **coarse-granularity parallelism**, with few processors (about two to four, typically), all sharing the main memory. Databases running on such machines usually do not attempt to partition a single query among the processors; instead, they run

each query on a single processor, **allowing multiple queries to run concurrently.** Thus, such systems support a higher throughput; that is, they allow a greater number of transactions to run per second.

- Machines with **fine-granularity parallelism** have a large number of processors, and database systems running on such machines attempt to **parallelize single tasks** (queries, for example) submitted by users.

✓ Client-Server Systems

- The server system satisfies requests generated by clients.
- Figure 5.15 shows the general structure of a client-server system.

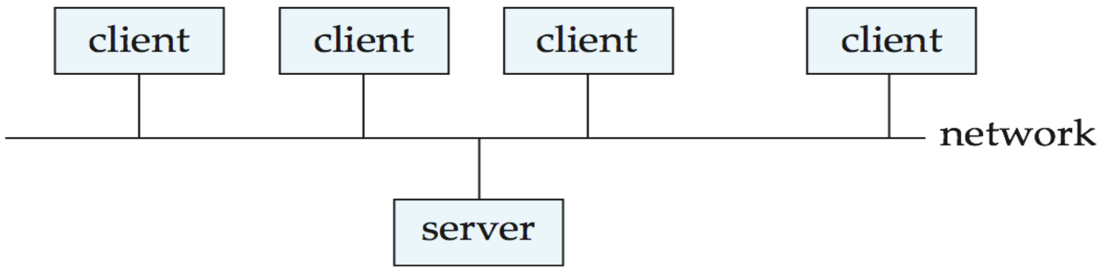


Figure 5.15:- General Structure of a Client-server system

- Database functionality can be divided into:
  - **Back-end**: manages access structures, query evaluation and optimization, concurrency control and recovery.
  - **Front-end**: consists of tools such as forms, report-writers, and graphical user interface facilities.
- The interface between the front-end and the back-end is through SQL or through an application program interface (figure 5.16).

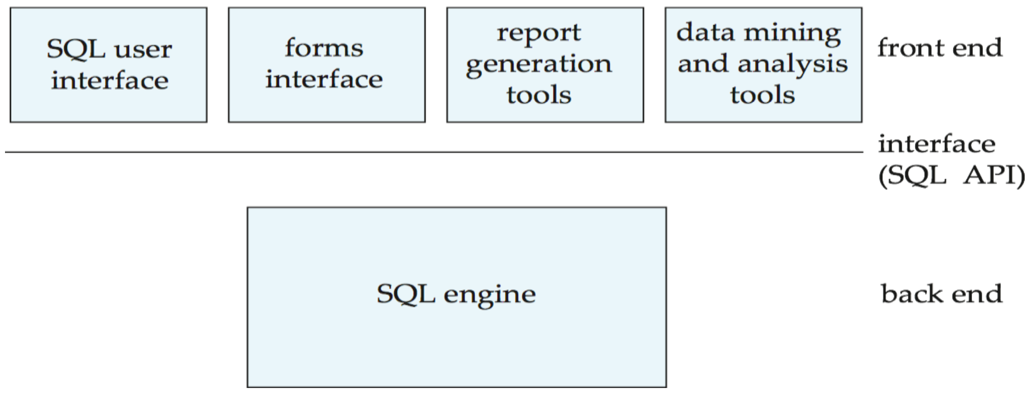


Figure 5.16: - Front-end and back-end functionality

- Standards such as **ODBC and JDBC** were developed to interface clients with servers. Any client that uses the ODBC or JDBC interface can connect to any server that provides the interface.
- Certain application programs, such as spreadsheets and statistical-analysis packages, use the client–server interface directly to access data from a back-end server.
- Systems that deal with large numbers of users adopt a three-tier architecture, where the front end is a Web browser that talks to an application server. The application server, in effect, acts as a client to the database server.
- Some transaction-processing systems provide a **transactional remote procedure call** interface to connect clients with a server.

➢ Server System Architecture

- Server systems can be broadly divided into two kinds:
  - **Transaction servers: -** Also called as query-server systems. They are widely used in relational database systems. Transaction server systems provide an interface to which clients can send requests to perform an action, in response to which they execute the action and send back results to the client.
  - **Data servers: -** They are used in object-oriented database systems. Data server system allows clients to interact with the servers by making requests to read or update data, in units such as files or pages. For example, file servers provide a file-system interface where clients can create, update, read, and delete files.

✓ Transaction Servers

- Also called **query server** systems or SQL server systems
  - Clients send requests to the server
  - Transactions are executed at the server
  - Results are shipped back to the client.
- Requests are specified in SQL, and communicated to the server through a remote procedure call (RPC) mechanism.

- Open Database Connectivity (ODBC) is a C language application program interface standard from Microsoft for connecting to a server, sending SQL requests, and receiving results.
- JDBC standard is similar to ODBC, for Java.
- A typical transaction server consists of multiple processes accessing data in shared memory (figure 5.17).

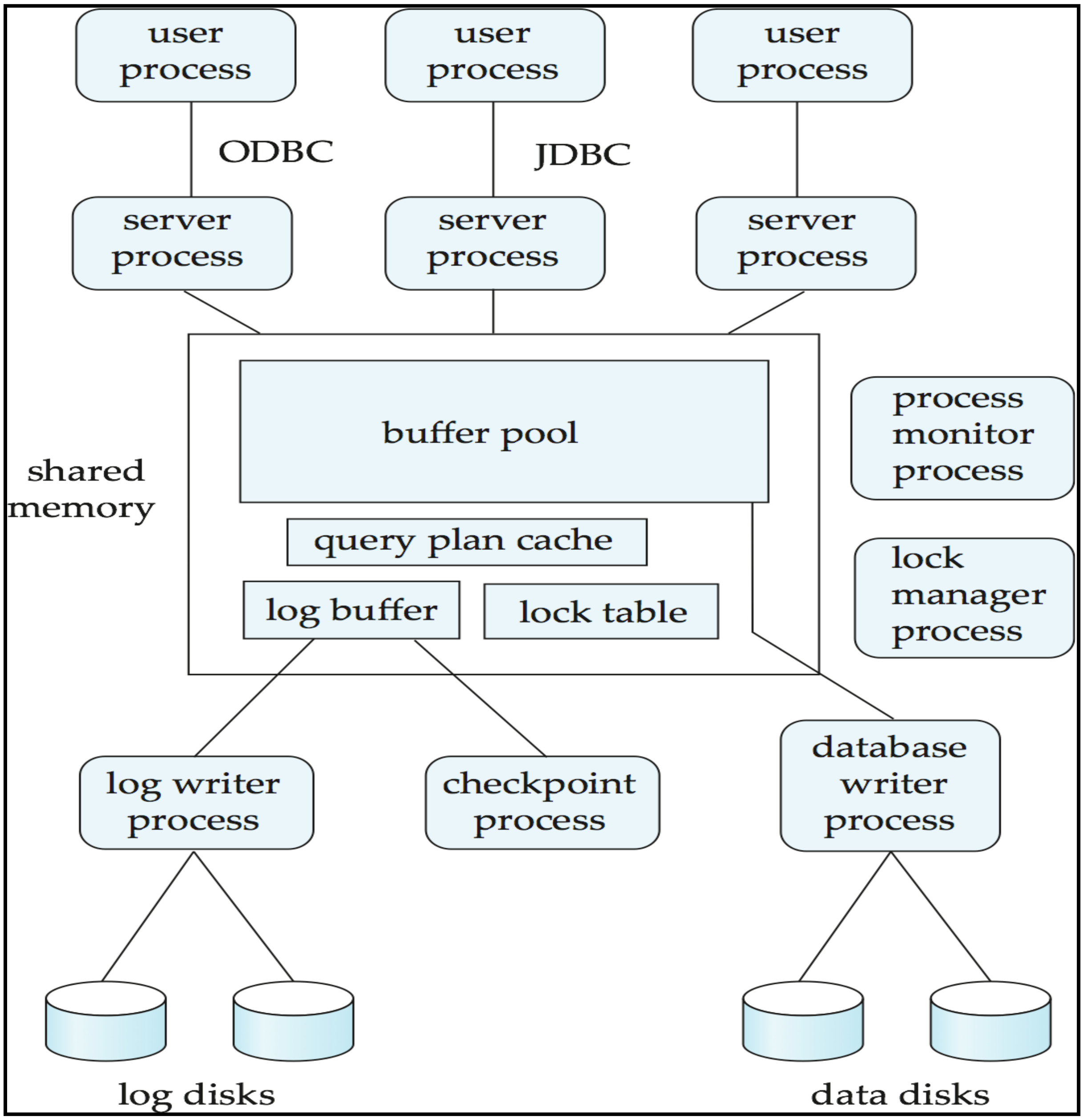


Figure 5.17:- Shared Memory and Process Structure

- A typical transaction server consists of multiple processes accessing data in shared memory. The processes that form part of the database system include:
  - **Server processes**
    - These receive user queries (transactions), execute them and send results back.
    - Processes may be **multithreaded**, allowing a single process to execute several user queries concurrently.
    - Many multithreaded server processes can be there.
  - **Lock manager process**
    - This process implements lock manager functionality, which includes lock grant, lock release.
  - **Database writer process**
    - It outputs modified buffer blocks to disks continually.
  - **Log writer process**
    - Server processes simply add log records to log record buffer.
    - Log writer process outputs log records to stable storage.
  - **Checkpoint process**
    - Performs periodic checkpoints.
  - **Process monitor process**
    - Monitors other processes, and takes recovery actions if any of the other processes fail.
    - E.g., aborting any transactions being executed by a server process and restarting it.
- Shared memory contains shared data such as:
  - Buffer pool
  - Lock table
  - Log buffer
  - Cached query plans (reused if same query submitted again)
- All database processes can access shared memory.

- To ensure that no two processes are accessing the same data structure at the same time, databases systems implement **mutual exclusion** using either
    - Operating system semaphores
    - Atomic instructions such as test-and-set
- To avoid overhead of interprocess communication (**IPC**) for lock request/grant, each database process operates directly on the lock table
    - instead of sending requests to lock manager process
- Lock manager process still used for deadlock detection.

✓ Data Servers

- Used in high-speed LANs, in cases where
    - The clients are comparable in processing power to the server
    - The tasks to be executed are compute intensive.
- Data are shipped to clients where processing is performed, and then shipped results back to the server.
- This architecture requires full back-end functionality at the clients.
- Used in many object-oriented database systems
- Issues:
    - **Page-Shipping versus Item-Shipping**
    - **Adaptive lock granularity**
    - **Data Caching**
    - **Lock Caching**
- **Page-shipping** versus **item-shipping**
    - The unit of communication for data can be of coarse granularity, such as a page, or fine granularity, such as a tuple
    - We use the term **item** to refer to both tuples and objects.
    - Fetching items even before they are requested is called prefetching.
    - Page shipping can be thought of as a form of prefetching if multiple items reside on a page.

- **Adaptive lock granularity**
  - Locks are usually granted by the server for the data items that it ships to the client machines.
  - A disadvantage of page shipping is that client machines may be granted locks —a lock on a page locks all items contained in the page.
  - Even if the client is not accessing some items in the page, it has acquired locks on all prefetched items.
  - Other client machines that require locks on those items may be blocked unnecessarily.
  - Techniques for lock **de-escalation** have been proposed where the server can request its clients to transfer back locks on prefetched items.
  - If the client machine does not need a prefetched item, it can transfer locks on the item back to the server, and the locks can then be allocated to other clients
- **Data Caching**
  - Data can be cached at client even in between transactions.
  - But check that data is up-to-date before it is used (**cache coherency**).
  - Check can be done when requesting lock on data item.
- **Lock Caching**
  - Locks can be retained by client system even in between transactions.
  - Transactions can acquire cached locks locally, without contacting server.
  - Server **calls back** locks from clients when it receives conflicting lock request. Client returns lock once no local transaction is using it.

➢ Parallel Systems

- In parallel processing, many operations are performed simultaneously, as opposed to serial processing, in which the computational steps are performed sequentially.
- Parallel database systems consist of multiple processors and multiple disks connected by a fast interconnection network.
- A **coarse-grain parallel** machine consists of a small number of powerful processors
- A **massively parallel** or **fine grain parallel** machine utilizes thousands of smaller processors.
- Two main performance measures:
  - **throughput** --- the number of tasks that can be completed in a given time interval
  - **response time** --- the amount of time it takes to complete a single task from the time it is submitted

- **Speedup**: Running a given task in less time by increasing the degree of parallelism is called speedup.
  - Measured by:

    $$\text{speedup} = \frac{\text{small system elapsed time}}{\text{large system elapsed time}}$$
  - Speedup is **linear** if equation equals N.
- **Scaleup**: Handling larger tasks by increasing the degree of parallelism.
  - N-times larger system used to perform N-times larger job
  - Measured by:

    $$\text{scaleup} = \frac{\text{small system small problem elapsed time}}{\text{big system big problem elapsed time}}$$
  - Scale up is **linear** if equation equals 1.

✓ Parallel Database Architecture

- There are several architectural models for parallel machines. Among the most prominent ones are those in Figure 5.18
  - **Shared memory** -- processors share a common memory
  - **Shared disk** -- processors share a common disk
  - **Shared nothing** -- processors share neither a common memory nor common disk
  - **Hierarchical** -- hybrid of the above architectures
- Some of the Parallel Database Architectures are as shown in following figure 5.18.

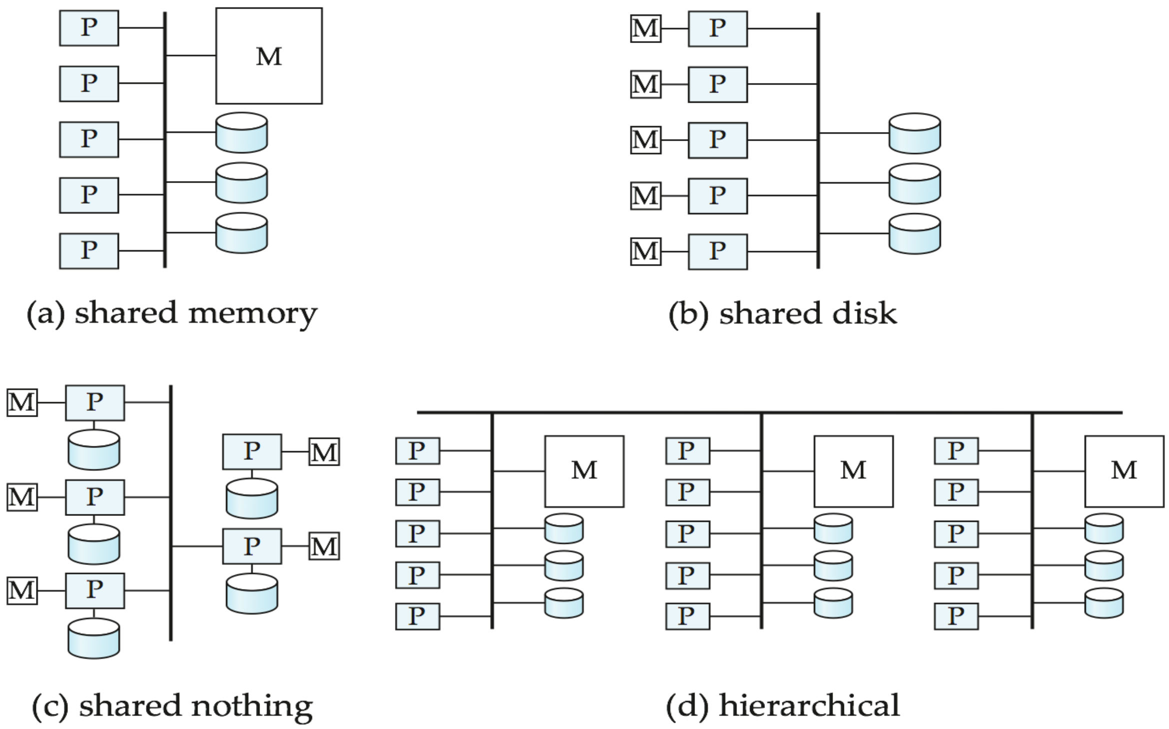


Figure 5.18:- Parallel Database Architectures

- **Shared Memory Architecture:**
  - Processors and disks have access to a common memory, typically via a bus or through an interconnection network.
  - Extremely efficient communication between processors — data in shared memory can be accessed by any processor without having to move it using software.

    - Downside – architecture is not scalable beyond 32 or 64 processors since the bus or the interconnection network becomes a bottleneck
    - Widely used for lower degrees of parallelism.
- **Shared Disk Architecture:**
    - All processors can directly access all disks via an interconnection network, but the processors have private memories.
        - The memory bus is not a bottleneck
        - Architecture provides a degree of **fault-tolerance** — if a processor fails, the other processors can take over its tasks since the database is resident on disks that are accessible from all processors.
    - Examples: IBM Sysplex and DEC clusters (now part of Compaq) running Rdb (now Oracle Rdb) were early commercial users
    - Downside: bottleneck now occurs at interconnection to the disk subsystem.
    - Shared-disk systems can scale to a somewhat larger number of processors, but communication between processors is slower.
- **Shared Nothing Architecture:**
    - Node consists of a processor, memory, and one or more disks. Processors at one node communicate with another processor at another node using an interconnection network. A node functions as the server for the data on the disk or disks the node owns.
    - Examples: Teradata, Tandem, Oracle-n CUBE
    - Data accessed from local disks (and local memory accesses) do not pass through interconnection network, thereby minimizing the interference of resource sharing.
    - Shared-nothing multiprocessors can be scaled up to thousands of processors without interference.
    - Main drawback: cost of communication and non-local disk access; sending data involves software interaction at both ends.

- **Hierarchical Architecture:**
  - Combines features of shared-memory, shared-disk, and shared-nothing architectures.
  - Top level is a shared-nothing architecture – nodes connected by an interconnection network, and do not share disks or memory with each other.
  - Each node of the system could be a shared-memory system with a few processors.
  - Alternatively, each node could be a shared-disk system, and each of the systems sharing a set of disks could be a shared-memory system.
  - Reduce the complexity of programming such systems by **distributed virtual-memory** architectures
    - Also called **non-uniform memory architecture (NUMA)**

➢ Distributed Systems

- In a **distributed database system**, the database is stored on several computers. The computers in a distributed system communicate with one another through various communication media, such as high-speed private networks or the Internet. They do not share main memory or disks. The computers in a distributed system may vary in size and function, ranging from workstations up to mainframe systems.
- Network interconnects the various machines.
- Data is shared by users on multiple machines.
- The computers in a distributed system are referred to by a number of different names, such as **sites** or **nodes**, depending on the context in which they are mentioned. We mainly use the term **site**, to emphasize the physical distribution of these systems.
- The general structure of a Distributed System appears in following figure 5.19.

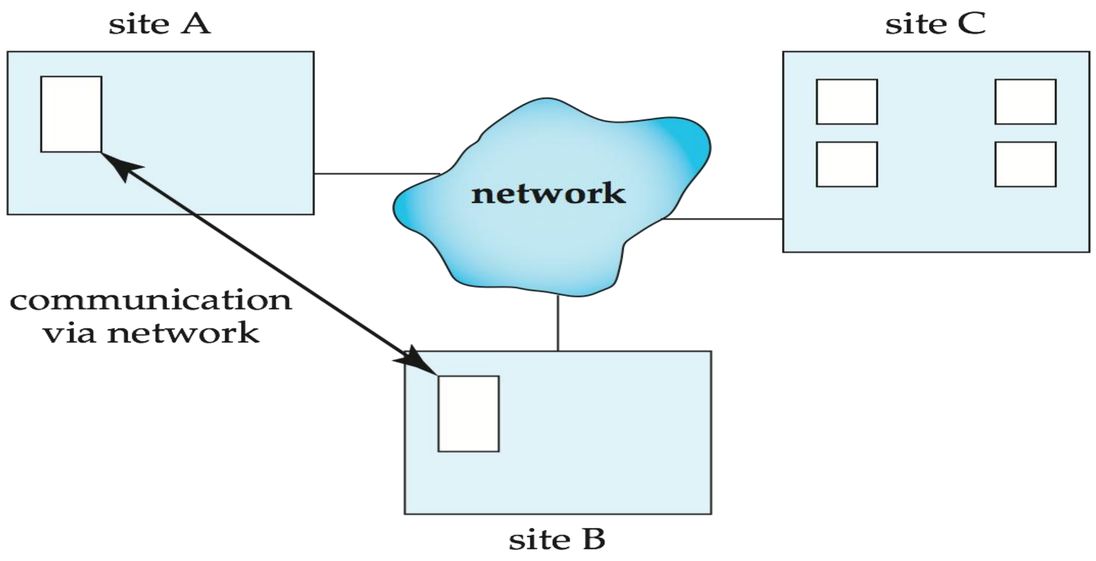


Figure 5.19:- A Distributed System

- **Homogeneous distributed databases**
  - Same software/schema on all sites, data may be partitioned among sites
  - Goal: provide a view of a single database, hiding details of distribution
- **Heterogeneous distributed databases**
  - Different software/schema on different sites
  - Goal: integrate existing databases to provide useful functionality
- Differentiate between local and global transactions
  - A **local transaction** accesses data in the single site at which the transaction was initiated.
  - A **global transaction** either accesses data in a site different from the one at which the transaction was initiated or accesses data in several different sites.
- There are several reasons for building distributed database systems including the following (i.e. various advantages of distributed database systems):-
  - **Sharing data** – users at one site able to access the data residing at some other sites.

- **Autonomy** – each site is able to retain a degree of control over data stored locally. The possibility of local autonomy is the major advantage of distributed database system.
- **Improved Availability and reliability**— data can be replicated at remote sites, and system can function even if a site fails. So the failure of a site does not necessarily imply the shutdown of a system.
- **Expansion**: - Distributed systems are more popular, hence, they can be expanded easily as compared to centralized system.

- Disadvantage: added complexity required to ensure proper coordination among sites.
  - Software development cost.
  - Greater potential for bugs.
  - Increased processing overhead.

✓ Implementation Issues for Distributed Database

- Atomicity of transactions is an important issue in building a distributed database system. If a transaction runs across two sites, unless the system designers are careful, it may commit at one site and abort at another, leading to an inconsistent state. Transaction commit protocols ensure such a situation cannot arise. The ***two-phase commit protocol*** (2PC) is the most widely used of these protocols.
- Concurrency control is another issue in a distributed database.
- Since a transaction may access data items at several sites, transaction managers at several sites may need to coordinate to implement concurrency control.
- When the tasks to be carried out are complex, involving multiple databases and/or multiple interactions with humans, coordination of the tasks and ensuring transaction properties for the tasks become more complicated. **Workflow management systems** are systems designed to help with carrying out such tasks.

- In case an organization has to choose between a distributed architecture and a centralized architecture for implementing an application, the system architect must balance the advantages against the disadvantages of distribution of data. The primary **disadvantage of distributed database systems** is the added complexity required to ensure proper coordination among the sites. This increased complexity takes various forms:
  - **Software-development cost: -** It is more difficult to implement a distributed database system; thus, it is more costly.
  - **Greater potential for bugs: -** Since the sites that constitute the distributed system operate in parallel, it is harder to ensure the correctness of algorithms especially operation during failures of part of the system, and recovery from failures. The potential exists for extremely subtle bugs.
  - **Increased processing overhead: -** The exchange of messages and the additional computation required to achieve intersite coordination are a form of overhead that does not arise in centralized systems.

Bibliography

1. Abraham Silberschatz, Henry F. Korth, S. Sudarshan, “Database System Concepts”, 6th Edition, McGraw-Hill Hill Education.
2. Abraham Silberschatz, Henry F. Korth, S. Sudarshan, “Database System Concepts”, 4th Edition, McGraw-Hill Hill Education.
3. Ramez Elmasri and Shamkant B. Navathe “Fundamentals of Database Systems”, 5th Edition, Pearson.
4. Express Learning, “Database Management Systems”, ITL Education Solutions Limited.
5. Archana Verma, “Database Management Systems”, GenNext Publication.
6. Dr. Rajiv Chopra, “Database Management Systems (DBMS) – A Practical Approach”, 5th Edition, S. Chand Technical
7. Tanmay Kasbe, “Database Management System Concepts – A Practical Approach”, First Edition, Educreation Publishing.
8. Mahesh Mali, “Database Management Systems”, Edition 2019, TechKnowledge Publications.
9. Rajendra Prasad Mahapatra, Govind Verma, “Database Management System”, Khanna Publishing.
10. Malay K. Pakhira, “Database Management System”, Eastern Economy Edition, PHI.
11. Sarika Gupta, Gaurav Gupta, “Database Management System”, Khanna Book Publishing Edition.
12. Riktesh Srivastava, Rajita Srivastava, “Relational Database Management System”, New Age International Publishers.
13. Peter Rob, Carlos Coronel, “Database System Concepts’, Cenage Learning, India Edition
14. Bipin C. Desai, “An Introduction to Database Systems”, Galgotia Publications.
15. G.K. Gupta, “Database Management Systems”, McGraw Hill Education.
16. Shio Kumar Singh, “Database Systems – Concepts, Design and Applications”, 2nd Edition, PEARSON.
17. S.D.Joshi, “Database Management System”, Tech-Max Publication.
18. R. Ramkrishnan , J. Gehrke, "Database Management Systems", 3rd Edition, McGraw-Hill
19. C. J. Date, “Introduction to Database Management Systems”, 8th Edition, Pearson
20. Atul Kahate, “Introduction to Database Management System”, 3rd Edition, Pearson.
21. Bharat Lohiya, “Database Systems”, Tenth Edition, Aditya Publication, Amravati.
22. Vijay Krishna Pallaw, “Database Management System”, 2nd, Asian Books Pvt. Ltd.
23. Database Management Systems, Database Management Systems.
24. Mrs. Jyoti G. Mante (Khurpade), Mrs. Smita M. Dandge, “Database Mangement System”, Nirali Prakashan.
25. Step by Step Database Systems (DBMS), Shiv Krupa Publications, Akola

26. Mrs. Sheetal Gujar –Takale, Mr. Sahil K. Shah, “Database Management System”, Nirali Prakashan.
27. Mrs. Jyoti G. Mante (Khurpade), U.S. Shirshetti, M.V. Salvi, K.S. Sakure, “Relational Database Management System”, Nirali Prakashan.
28. Seema Kedar, Rakesh Shirsath, “Database Management Systems”, Technical Publications.
29. Pankaj B. Brahmankar, “Database Management Systems”, Tech-Max Publications, Pune.
30. Imran Saeed, Tasleem Mustafa, Tariq Mahmood, Ahsan Raza Sattar, “A Fundamental Study of Database Management Systems”, 3rd Edition, IT Series Publication.
31. Database Management Systems Lecture Notes, Malla Reddy College of Engineering and Technology, Secunderabad.
32. Dr. Satinder Bal Gupta, Aditya Mittal, “Introduction to Database Management System, University Science Press.
33. E-Notes BCS 41/ BCA 41 on “Database Management System”, Thiruvalluvar University.
34. Bighnaraj Naik, Digital Notes on “Relational Database Management System”, VSSUT, Burla.
35. Viren Sir, Relational database Management System”, Adarsh Institute of Technolgoyt (Poly), VITA.
36. Sitansu S. Mitra, “Principles of Relational Database Systems”, Prentice Hall.
37. Neeraj Sharma, Liviu Perniu, Raul F. Chong, Abhishek Iyer, Chaitali Nandan, Adi-Cristina Mitea, Mallarswami Nonvinkere, Mirela Danubianu, “Database Fundamentals”, First Edition, DB2 On Campus Book Series.
38. Database Management System, Vidyavahini First Grade College, Tumkur.
39. Bhavna Sangamnerkar, Revised by: Shiv Kishor Sharma, “Database Management System”, Think Tanks Biyani Group of Colleges.
40. Tibor Radvanyi, “Database Management Systems”.
41. Ramon A. Mata-Toledo, Pauline K. Cushman, “Fundamentals of Relational Databases”, Schaum’s Outlies.
42. P.S. Gill, “Database Management Systems”, 2nd Edition, Dreamtech Press, WILEY

Bibliography

Web Resources:-

https://www.exploredatabase.com/2016/04/database-transaction-states-in-dbms.html
https://www.javatpoint.com/dbms-log-based-recovery
https://www.geeksforgeeks.org/difference-between-deferred-update-and-immediate-update/