SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT I I - DISTRIBUTED DATABASE AND INFORMATION SYSTEMS - SCSA3008

Unit:2

Overview of security techniques - Cryptographic algorithms - Digital signatures - Distributed Concurrency Control - Serializability theory - Taxonomy of concurrency control mechanisms - Distributed deadlocks – Distributed Database Recovery - Distributed Data Security - Web data management - Database Interoperability.

Security

Security refers to protecting and securing computers and their related data, networks, software, hardware from unauthorized access, misuse, theft, information loss, and other security issues. Technology is growing day by day and the entire world is in its grasp. With the use of this growing technology, invaders, hackers and thieves are trying to harm our computer's security for monetary gains, recognition purposes, ransom demands, bullying others, invading into other businesses, organizations, etc. In order to protect our system from all these risks, computer security is important.

Types of security

Security can be classified into four types:

1. Cyber Security: Cyber security means securing our computers, electronic devices, networks , programs, systems from cyber attacks. Cyber attacks are those attacks that happen when our system is connected to the Internet.

2. Information Security: Information security means protecting our system's information from theft, illegal use and piracy from unauthorized use.

3. Application Security: Application security means securing our applications and data so that they don't get hacked and also the databases of the applications remain safe and private to the owner itself so that user's data remains confidential.

4. Network Security: Network security means securing a network and protecting the user's information about who is connected through that network

Types of cyber attack

1. Denial of service attack or DOS: A denial of service attack is a kind of cyber attack in which the attackers disrupt the services of the particular network by sending infinite requests and temporary or permanently making the network or machine resources unavailable to the intended audience.

2. Backdoor: In a backdoor attack, malware, trojan horse or virus gets installed in our system and start affecting it's security along with the main file.

3.Eavesdropping: Eavesdropping refers to secretly listening to someone's talk without their permission or knowledge.

4. Phishing: Phishing, a user is tricked by the attacker who gains the trust of the user or acts as if he is a genuine person and then steals the information by ditching. Not only attackers but some certain websites that seem to be genuine, but actually they are fraud sites. These sites trick the users and they end up giving their personal information such as login details or bank details or card number etc. Phishing is of many types: Voice phishing, text phishing etc.

5. Spoofing: Spoofing is the act of masquerading as a valid entity through falsification of data(such as an IP address or username), in order to gain access to information or resources that one is otherwise unauthorized to obtain. Spoofing is of several types- email spoofing, IP address spoofing, MAC spoofing , biometric spoofing etc.

6. Malware: Malware is made up of two terms: Malicious + Software = Malware. Malware intrudes into the system and is designed to damage our computers. Different types of malware are adware, spyware, ransomware, Trojan horse, etc.

7. Social engineering: Social engineering attack involves manipulating users psychologically and extracting confidential or sensitive data from them by gaining their trust. The attacker generally exploits the trust of people or users by relying on their cognitive basis.

Security Techniques

1.Interfaces are exposed: Distributed systems are composed of processes that offer services or share information. Their communication interfaces are necessarily open (to allow new clients to access them) – an attacker can send a message to any interface.

2.Networks are insecure: For example, message sources can be falsified.

3.Limit the lifetime and scope of each secret: When a secret key is first generated we can be confident that it has not been compromised. The use of secrets such as passwords and shared secret keys should be time-limited, and sharing should be restricted.

4.Algorithms and program code are available to attackers: Secret encryption algorithms are totally inadequate for today's large-scale network environments. Best practice is to publish the algorithms used for encryption and authentication, relying only on the secrecy of cryptographic keys. This helps to ensure that the algorithms are strong by throwing them open to scrutiny by third parties.

5.Attackers may have access to large resources: The cost of computing power is rapidly decreasing. We should assume that attackers will have access to the largest and most powerful computers projected in the lifetime of a system, then add a few orders of magnitude to allow for unexpected developments.

6.Minimize the trusted base: The portions of a system that are responsible for the implementation of its security, and all the hardware and software components upon which they rely, have to be trusted – this is often referred to as the trusted computing base. Any defect or programming error in this trusted base can produce security weaknesses, so we should aim to minimize its size. For example, application programs should not be trusted to protect data from their users.

Cryptography Algorithms

- **Encryption** is the process of encoding a message in such a way as to hide its contents. Modern cryptography includes several secure algorithms for encrypting and decrypting messages. They are all based on the use of secrets called **keys**.
- A cryptographic key is a parameter used in an encryption algorithm in such a way that the encryption cannot be reversed without knowledge of the key.
- The first uses shared secret keys – the sender and the recipient must share a knowledge of the key and it must not be revealed to anyone else. The second class of encryption algorithms uses public/private key pairs. Here the sender of a message uses a public key – one that has already been published by the recipient – to encrypt the message. The recipient uses a corresponding private key to decrypt the message.

Uses of cryptography

Cryptography plays three major roles in the implementation of secure systems.

Secrecy and integrity-Cryptography is used to maintain the secrecy and integrity of information whenever it is exposed to potential attacks – for example, during transmission across networks that are vulnerable to eavesdropping and message tampering. This use of cryptography corresponds to its traditional role in military and

Authentication: Cryptography is used in support of mechanisms for authenticating communication between pairs of principals. A principal who decrypts a message successfully using a particular key can assume that the message is authentic if it contains a correct checksum or some other expected value.

Algorithms

1. **Symmetric Key Cryptography:** It is an encryption system where the sender and receiver of message use a single common key to encrypt and decrypt messages. Symmetric Key Systems are faster and simpler but the problem is that sender and receiver have to somehow exchange key in a secure manner. The most popular symmetric key cryptography system is Data Encryption System(DES).
2. **Hash Functions:** There is no usage of any key in this algorithm. A hash value with fixed length is calculated as per the plain text which makes it impossible for contents of plain text to be recovered. Many operating systems use hash functions to encrypt passwords.
3. **Asymmetric Key Cryptography:** Under this system a pair of keys is used to encrypt and decrypt information. A public key is used for encryption and a private key is used for decryption. Public key and Private Key are different. Even if the public key is known by everyone the intended receiver can only decode it because he alone knows the private key.

Digital Signatures

A **digital signature** is a mathematical scheme for verifying the authenticity of digital messages or documents. A valid digital signature, where the prerequisites are satisfied, gives a recipient very high confidence that the message was created by a known sender, and that the message was not altered in transit.

Digital signatures are a standard element of most cryptographic protocol suites, and are commonly used for software distribution, financial transactions, contract management software, and in other cases where it is important to detect forgery or tampering.

Digital signatures are often used to implement electronic signatures, which includes any electronic data that carries the intent of a signature, but not all electronic signatures use digital signatures.

Digital signatures employ asymmetric cryptography. In many instances, they provide a layer of validation and security to messages sent through a non-secure channel: Properly implemented, a digital signature gives the receiver reason to believe the message was sent by the claimed sender. Digital signatures are equivalent to traditional handwritten signatures in many respects, but properly implemented digital signatures are more difficult to forge than the handwritten type. Digital signature schemes, in the sense used here, are cryptographically based, and must be implemented properly to be effective. They can also provide non-repudiation, meaning that the signer cannot successfully claim they did not sign a message, while also claiming their private key remains secret. Further, some non-repudiation schemes offer a timestamp for the digital signature, so that even if the private key is exposed, the signature is valid. Digitally signed messages may be anything representable as a bitstring: examples include electronic mail, contracts, or a message sent via some other cryptographic protocol.

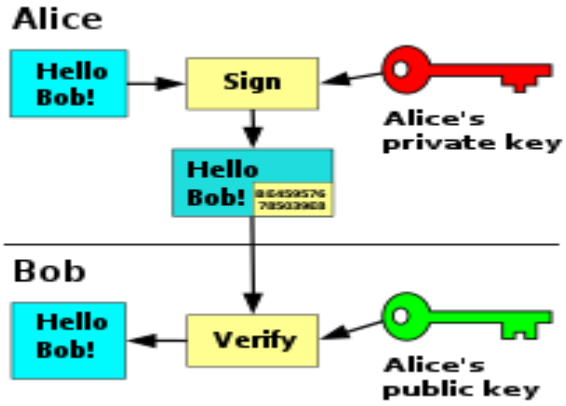

Fig 2.1 Digital Signature

Alice signs a message—"Hello Bob!"—by appending a signature computed from the message and her private key. Bob receives both the message and signature. He uses Alice's public key to verify the authenticity of the signed message shown in Fig 2.1.

Distributed Concurrency Control

Concurrency controlling techniques ensure that multiple transactions are executed simultaneously while maintaining the ACID properties of the transactions and serializability in the schedules.

Concurrency control is provided in a database to:

- (i) enforce isolation among transactions.
- (ii) preserve database consistency through consistency preserving execution of transactions.
- (iii) resolve read-write and write-read conflicts.

Locking Based Concurrency Control Protocols

Locking-based concurrency control protocols use the concept of locking data items. A **lock** is a variable associated with a data item that determines whether read/write operations can be performed on that data item. Generally, a lock compatibility matrix is used which states whether a data item can be locked by two transactions at the same time. The operation's access request is decided based on the compatibility of lock modes—read lock is compatible with another read lock, a write lock is not compatible with either a read or a write lock.

Locking-based concurrency control systems can use either one-phase or two-phase locking protocols.

One-phase Locking Protocol

In this method, each transaction locks an item before use and releases the lock as soon as it has finished using it. This locking method provides for maximum concurrency but does not always enforce serializability.

Two-phase Locking Protocol

In this method, all locking operations precede the first lock-release or unlock operation. The transaction comprise of two phases. In the first phase, a transaction only acquires all the locks it needs and do not release any lock. This is called the expanding or the **growing phase**. In the

second phase, the transaction releases the locks and cannot request any new locks. This is called the **shrinking phase**. The fundamental decision in distributed locking-based concurrency control algorithms is where and how the locks are maintained (usually called a lock table).

Centralized 2PL

The 2PL algorithm can easily be extended to the distributed DBMS environment by delegating lock management responsibility to a single site. This means that only one of the sites has a lock manager; the transaction managers at the other sites communicate with it to obtain locks. This approach is also known as the **primary site 2PL algorithm**

This communication is between the *coordinating*TM, the lock manager at the central site, and the data processors (DP) at the otherparticipating sites. The participating sites are those that store the data items on whichthe operatio\n is to be carried out.These algorithms use a 5-tuple for the operation they perform: *Op : _Type ={BT,R,W,A,C}*.The transaction manager (C2PL-TM) algorithm is written as a process thatruns forever and waits until a message arrives from either an application (with atransaction operation) or from a lock manager, or from a data processor shown in fig 2.2. The lockmanager (C2PL-LM) and data processor (DP) algorithms are written as proceduresthat are called when needed.

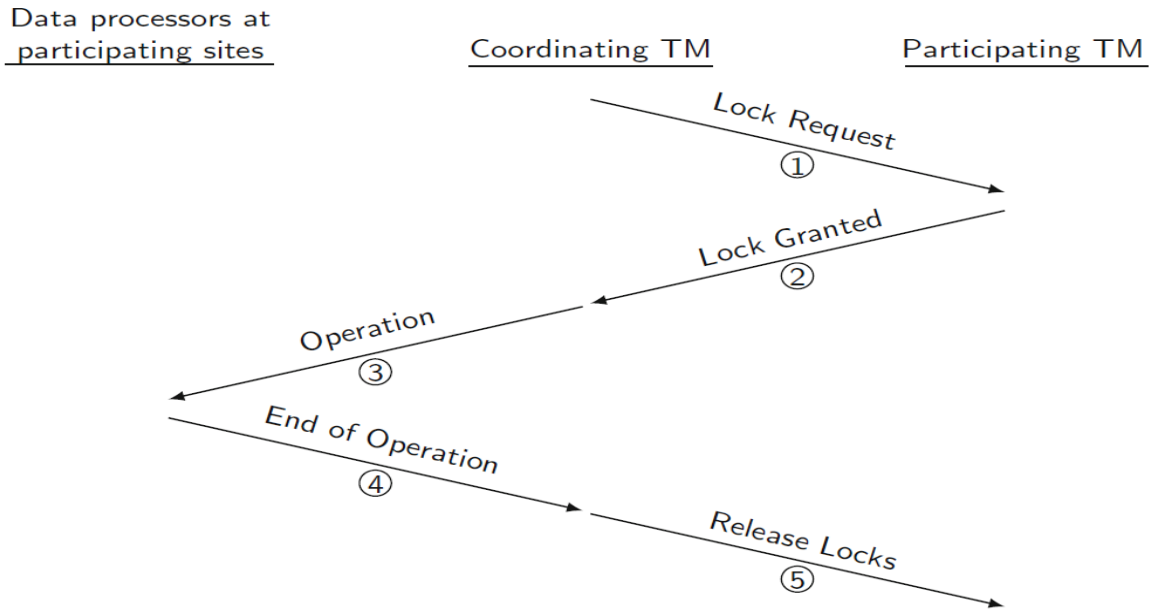

Fig 2.2 Centralized 2PL

Distributed 2PL

Distributed 2PL (D2PL) requires the availability of lock managers at each site. Thecommunication between cooperating sites that execute a transaction according tothe distributed 2PL protocol.The distributed 2PL transaction management algorithm is similar to the C2PLTM,with two major modifications. The messages that are sent to the central sitelock manager in C2PL-TM are sent to the lock managers at all participating sites inD2PL-TM. The second difference is that the operations are not passed to the dataprocessors by the coordinating transaction manager, but by the participating lockmanagers. This means that the coordinating transaction manager does not wait fora "lock request granted" message shown in Fig 2.3.

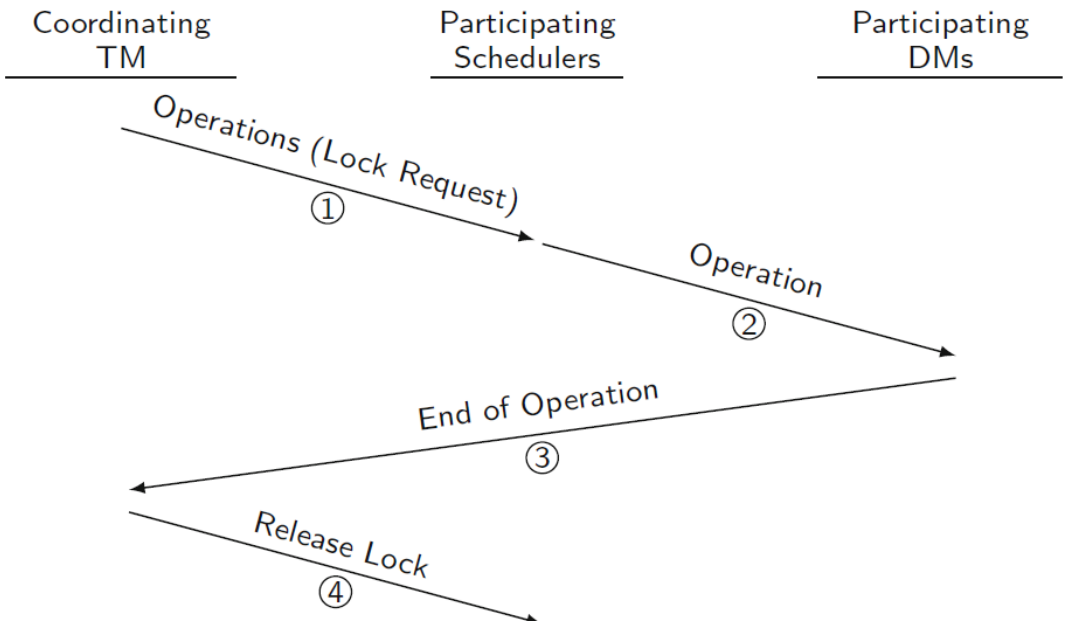

Fig 2.3 Distributed 2PL

Distributed Deadlock Management

Locking-based concurrency control algorithms may cause deadlocks; in the caseof distributed DBMSs, these could be *distributed* (or *global*) *deadlocks* due totransactions executing at different sites waiting for each other. Deadlock detectionand resolution is the most popular approach to managing deadlocks in the distributedsetting. The wait-for graph (WFG) can be useful for detecting deadlocks; this is adirected graph whose vertices are active transactions with an edge from *Ti*to *Tj*ifan operation in *Ti*is waiting to access a data item that is currently locked in anincompatible mode by an operation in *Tj*. However, the formation of the WFGis more complicated in a distributed setting due to the distributed execution oftransactions. Therefore, it is not sufficient for each site to form a *local wait-forgraph* (LWFG) and check it; it is also necessary to form a *global wait-for graph*(GWFG), which is the union of all the LWFGs, and check it for cycles.

There are threefundamental methods of detecting distributed deadlocks, referred as *centralized, distributed*, and *hierarchical deadlock detection*.

Centralized Deadlock Detection

In the centralized deadlock detection approach, one site is designated as thedeadlock detector for the entire system. Periodically, each lock manager transmits itsLWFG to the deadlock detector, which then forms the GWFG and looks for cycles init. The lock managers need only send changes in their graphs (i.e., the newly createdor deleted edges) to the deadlock detector. The length of intervals for transmittingthis information is a system design decision: the smaller the interval, the smallerthe delays due to undetected deadlocks, but the higher the deadlock detection andcommunication overhead.

Hierarchical Deadlock Detection

Deadlocks that are local to a single site would bedetected at that site using the LWFG. Each site also sends its LWFG to the deadlockdetector at the next level. Thus, distributed deadlocks involving two or more siteswould be detected by a deadlock detector in the next lowest level that has controlover these sites. For example, a deadlock at site 1 would be detected by the localdeadlock detector (*DD*) at site 1 (denoted *DD*21, 2 for level 2, 1 for site 1). If,however, the

deadlock involves sites 1 and 2, then *DD*11 detects it. Finally, if t thedeadlock involves sites 1 and 4, *DD*0*x* detects it, where *x* is one of 1, 2, 3, or 4. The hierarchical deadlock detection method reduces the dependence on thecentral site, thus reducing the communication cost. It is, however, considerably morecomplicated to implement and would involve nontrivial modifications to the lockand transaction manager algorithms.

Distributed Deadlock Detection

Distributed deadlock detection algorithms delegate the responsibility of detecting. The LWFG at each site is formed and is modified as follows:

1. Since each site receives the potential deadlock cycles from other sites, theseedges are added to the LWFGs.

2. The edges in the LWFG that show that local transactions are waiting fortransactions at other sites are joined with edges in the LWFGs depicting thatremote transactions are waiting for local ones.Thus, as in the hierarchical deadlock detection, thereare local deadlock detectors at each site that communicate their LWFGs with oneanother.

Timestamp-Based Algorithms

Timestamp-based concurrency control algorithms select, a priori, a serializationorder and execute transactions accordingly. To establish this ordering, the transactionmanager assigns each transaction *Ti*a unique *timestamp*, *ts(Ti)*, at its initiation. To maintainuniqueness, each site appends its own identifier to the counter value. Thus thetimestamp is a two-tuple of the form _local counter value, site identifier_. Note thatthe site identifier is appended in the least significant position. Hence it serves onlyto order the timestamps of two transactions that might have been assigned the samelocal counter value. the transaction manager is responsible for assigninga timestamp to each new transaction and attaching this timestamp to eachdatabase operation that it passes on to the scheduler.

Basic TO Algorithm

In the basic TO algorithm the coordinating TM assigns the timestamp to eachtransaction *Ti*[*ts(Ti)*], determines the sites where each data item is stored, and sends the relevant operations to these sites.

TO Rule Given two conflicting operations *Oij*and *Okl*belonging, respectively, totransactions *Ti*and *Tk*, *Oij*is executed before *Okl*if and only if *ts(Ti) <ts(Tk)*. Inthis case *Ti*is said to be the *older* transaction and *Tk* is said to be the *younger* one.A scheduler that enforces the TO rule checks each new operation againstconflicting operations that have already been scheduled. If the new operationbelongs to a transaction that is younger than all the conflicting ones that have already

been scheduled, the operation is accepted; otherwise, it is rejected, causing the entiretransaction to restart with a *new* timestamp.To facilitate checking of the TO Rule, each data item *x* is assigned twotimestamps: a *read timestamp* [*rts(x)*], which is the largest of the timestamps ofthe transactions that have read *x*, and a *write timestamp* [*wts(x)*], which is thelargest of the timestamps of the transactions that have written (updated) *x*. It isnow sufficient to compare the timestamp of an operation with the read and writetimestamps of the data item that it wants to access to determine if any transactionwith a larger timestamp has already accessed the same data item.When an operation is rejected by a scheduler, the corresponding transaction isrestarted by the transaction manager with a new timestamp. This ensures that thetransaction has a chance to execute in its next try.

Optimistic Concurrency Control Algorithm

In systems with low conflict rates, the task of validating every transaction for serializability may lower performance. In these cases, the test for serializability is postponed to just before commit. Since the conflict rate is low, the probability of aborting transactions which are not serializable is also low. This approach is called optimistic concurrency control technique.

In this approach, a transaction's life cycle is divided into the following three phases −

- **Execution Phase** − A transaction fetches data items to memory and performs operations upon them.
- **Validation Phase** − A transaction performs checks to ensure that committing its changes to the database passes serializability test.
- **Commit Phase** − A transaction writes back modified data item in memory to the disk.

This algorithm uses three rules to enforce serializability in validation phase −

Rule 1 − Given two transactions T_i and T_j, if T_i is reading the data item which T_j is writing, then T_i's execution phase cannot overlap with T_j's commit phase. T_j can commit only after T_i has finished execution.

According to this rule, a transaction must be validated locally at all sites when it executes. If a transaction is found to be invalid at any site, it is aborted. Local validation guarantees that the transaction maintains serializability at the sites where it has been executed. After a transaction passes local validation test, it is globally validated.

Rule 2 − Given two transactions T_i and T_j, if T_i is writing the data item that T_j is reading, then T_i's commit phase cannot overlap with T_j's execution phase. T_j can start executing only after T_i has already committed.

According to this rule, after a transaction passes local validation test, it should be globally validated. Global validation ensures that if two conflicting transactions run together at more than one site, they should commit in the same relative order at all the sites they run together. This may require a transaction to wait for the other conflicting transaction, after validation before commit. This requirement makes the algorithm less optimistic since a transaction may not be able to commit as soon as it is validated at a site.

Rule 3 − Given two transactions T_i and T_j, if T_i is writing the data item which T_j is also writing, then T_i's commit phase cannot overlap with T_j's commit phase. T_j can start to commit only after T_i has already committed.

Serializability in distributed database

In a system with a number of simultaneous transactions, a **schedule** is the total order of execution of operations. Given a schedule S comprising of n transactions, say T1, T2, T3……..Tn; for any transaction Ti, the operations in Ti must execute as laid down in the schedule S.

Types of Schedules

There are two types of schedules −

- **Serial Schedules** − In a serial schedule, at any point of time, only one transaction is active, i.e. there is no overlapping of transactions. This is depicted in the following graph in Fig 2.4.

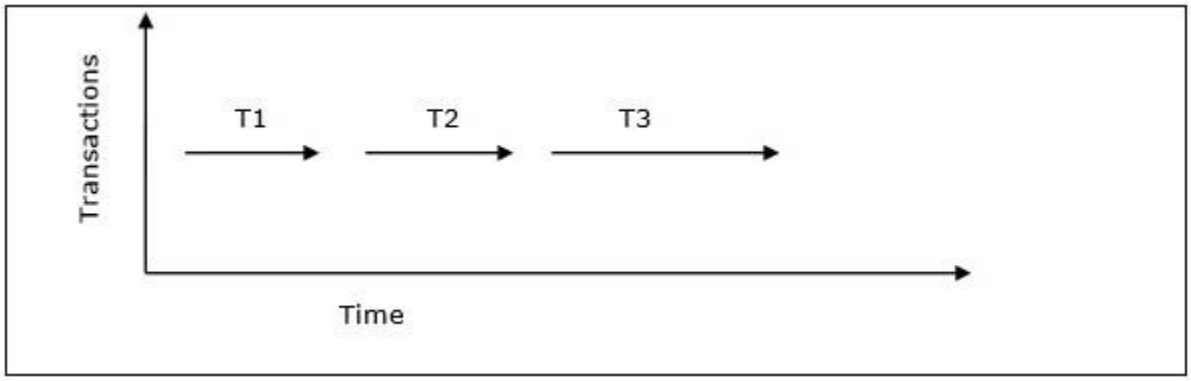

Fig 2.4 Serial Schedules

- **Parallel Schedules** − In parallel schedules, more than one transactions are active simultaneously, i.e. the transactions contain operations that overlap at time. This is depicted in the following graph shown in Fig 2.5

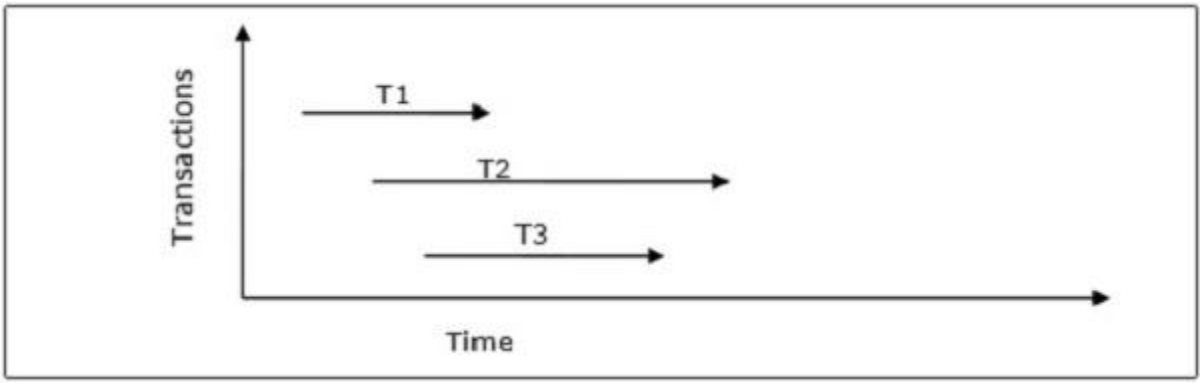

Fig 2.5 Parallel Schedules

Conflicts in Schedules

In a schedule comprising of multiple transactions, a **conflict** occurs when two active transactions perform non-compatible operations. Two operations are said to be in conflict, when all of the following three conditions exists simultaneously −

- The two operations are parts of different transactions.

- Both the operations access the same data item.
- At least one of the operations is a write_item() operation, i.e. it tries to modify the data item.

Serializability

A **serializable schedule** of 'n' transactions is a parallel schedule which is equivalent to a serial schedule comprising of the same 'n' transactions. A serializable schedule contains the correctness of serial schedule while ascertaining better CPU utilization of parallel schedule.

Equivalence of Schedules

Equivalence of two schedules can be of the following types −

- **Result equivalence** − Two schedules producing identical results are said to be result equivalent.
- **View equivalence** − Two schedules that perform similar action in a similar manner are said to be view equivalent.
- **Conflict equivalence** − Two schedules are said to be conflict equivalent if both contain the same set of transactions and has the same order of conflicting pairs of operations.

Serial schedules have less resource utilization and low throughput. To improve it, two are more transactions are run concurrently. But concurrency of transactions may lead to inconsistency in database. To avoid this, we need to check whether these concurrent schedules are serializable or not.

Conflict Serializable: A schedule is called conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations.

Conflicting operations: Two operations are said to be conflicting if all conditions satisfy:

- They belong to different transaction
- They operation on same data item
- At Least one of them is a write operation

Example: –

- **Conflicting** operations pair ($R_1(A)$, $W_2(A)$) because they belong to two different transactions on same data item A and one of them is write operation.
- Similarly, ($W_1(A)$, $W_2(A)$) and ($W_1(A)$, $R_2(A)$) pairs are also **conflicting**.
- On the other hand, ($R_1(A)$, $W_2(B)$) pair is **non-conflicting**because they operate on different data item.
- Similarly, (($W_1(A)$, $W_2(B)$) pair is **non-conflicting.**

Consider the following schedule:

S1: $R_1(A)$, $W_1(A)$, $R_2(A)$, $W_2(A)$, $R_1(B)$, $W_1(B)$, $R_2(B)$, $W_2(B)$

If O_i and O_j are two operations in a transaction and $O_i<O_j$ (O_i is executed before O_j), same order will follow in schedule as well. Using this property, we can get two transactions of schedule S1 as:

T1: $R_1(A)$, $W_1(A)$, $R_1(B)$, $W_1(B)$

T2: $R_2(A)$, $W_2(A)$, $R_2(B)$, $W_2(B)$

Possible Serial Schedules are: T1->T2 or T2->T1

-> **Swapping non-conflicting operation**s $R_2(A)$ and $R_1(B)$ in S1, the schedule becomes,

S11: $R_1(A)$, $W_1(A)$, $R_1(B)$, **$W_2(A)$,** $R_2(A)$, **$W_1(B)$,** $R_2(B)$, $W_2(B)$

-> Similarly, s**wapping non-conflicting operations** $W_2(A)$ and $W_1(B)$ in S11, the schedule becomes,

S12: $R_1(A)$, $W_1(A)$, $R_1(B)$, $W_1(B)$, $R_2(A)$, $W_2(A)$, $R_2(B)$, $W_2(B)$

S12 is a serial schedule in which all operations of T1 are performed before starting any operation of T2. Since S has been transformed into a serial schedule S12 by swapping non-conflicting operations of S1, S1 is conflict serializable.

Let us take another Schedule:

S2: $R_2(A)$, $W_2(A)$, $R_1(A)$, $W_1(A)$, $R_1(B)$, $W_1(B)$, $R_2(B)$, $W_2(B)$

Two transactions will be:

T1: $R_1(A)$, $W_1(A)$, $R_1(B)$, $W_1(B)$

T2: $R_2(A)$, $W_2(A)$, $R_2(B)$, $W_2(B)$

Possible Serial Schedules are: T1->T2 or T2->T1

Original Schedule is:

S2: $R_2(A)$, $W_2(A)$, **$R_1(A)$,** $W_1(A)$, $R_1(B)$, $W_1(B)$, **$R_2(B)$,** $W_2(B)$

Swapping non-conflicting operations $R_1(A)$ and $R_2(B)$ in S2, the schedule becomes,

S21: $R_2(A)$, $W_2(A)$, $R_2(B)$, **$W_1(A)$,** $R_1(B)$, $W_1(B)$, $R_1(A)$, **$W_2(B)$**

Similarly, swapping non-conflicting operations $W_1(A)$ and $W_2(B)$ in S21, the schedule becomes,

S22: $R_2(A)$, $W_2(A)$, $R_2(B)$, $W_2(B)$, $R_1(B)$, $W_1(B)$, $R_1(A)$, $W_1(A)$

In schedule S22, all operations of T2 are performed first, but operations of T1 are not in order (order should be $R_1(A)$, $W_1(A)$, $R_1(B)$, $W_1(B)$). So S2 is not conflict serializable.

Conflict Equivalent: Two schedules are said to be conflict equivalent when one can be transformed to another by swapping non-conflicting operations. In the example discussed above, S11 is conflict equivalent to S1 (S1 can be converted to S11 by swapping non-conflicting operations). Similarly, S11 is conflict equivalent to S12 and so on.

Note 1: *Although S2 is not conflict serializable, but still it is conflict equivalent to S21 and S21 because S2 can be converted to S21 and S22 by swapping non-conflicting operations.*

Note 2: *The schedule which is conflict serializable is always conflict equivalent to one of the serial schedule. S1 schedule discussed above (which is conflict serializable) is equivalent to serial schedule (T1->T2).*

Distributed deadlocks

Deadlock is a state of a database system having two or more transactions, when each transaction is waiting for a data item that is being locked by some other transaction. A deadlock can be indicated by a cycle in the wait-for-graph. This is a directed graph in which the vertices denote transactions and the edges denote waits for data items.

For example, in the following wait-for-graph, transaction T1 is waiting for data item X which is locked by T3. T3 is waiting for Y which is locked by T2 and T2 is waiting for Z which is locked by T1. Hence, a waiting cycle is formed, and none of the transactions can proceed executing shown in Fig 2.6.

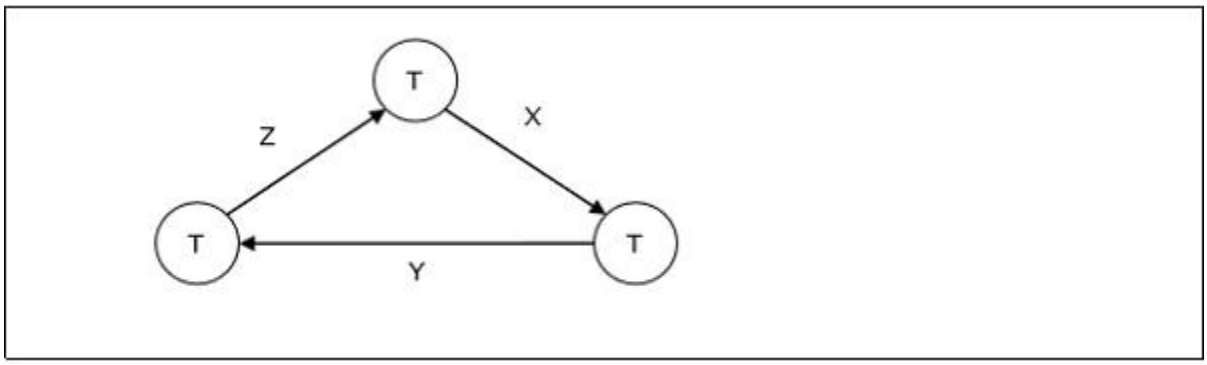

Fig 2.6 Deadlocks

Deadlock Handling in Centralized Systems

There are three classical approaches for deadlock handling, namely −

- Deadlock prevention.
- Deadlock avoidance.
- Deadlock detection and removal.

All of the three approaches can be incorporated in both a centralized and a distributed database system.

Deadlock Prevention

The deadlock prevention approach does not allow any transaction to acquire locks that will lead to deadlocks. The convention is that when more than one transactions request for locking the same data item, only one of them is granted the lock.

One of the most popular deadlock prevention methods is pre-acquisition of all the locks. In this method, a transaction acquires all the locks before starting to execute and retains the locks for the entire duration of transaction. If another transaction needs any of the already acquired locks, it has to wait until all the locks it needs are available. Using this approach, the system is prevented from being deadlocked since none of the waiting transactions are holding any lock.

Deadlock Avoidance

The deadlock avoidance approach handles deadlocks before they occur. It analyzes the transactions and the locks to determine whether or not waiting leads to a deadlock.

The method can be briefly stated as follows. Transactions start executing and request data items that they need to lock. The lock manager checks whether the lock is available. If it is available, the lock manager allocates the data item and the transaction acquires the lock. However, if the item is locked by some other transaction in incompatible mode, the lock manager runs an algorithm to test whether keeping the transaction in waiting state will cause a deadlock or not. Accordingly, the algorithm decides whether the transaction can wait or one of the transactions should be aborted.

There are two algorithms for this purpose, namely **wait-die** and **wound-wait**. Let us assume that there are two transactions, T1 and T2, where T1 tries to lock a data item which is already locked by T2. The algorithms are as follows −

- **Wait-Die** − If T1 is older than T2, T1 is allowed to wait. Otherwise, if T1 is younger than T2, T1 is aborted and later restarted.
- **Wound-Wait** − If T1 is older than T2, T2 is aborted and later restarted. Otherwise, if T1 is younger than T2, T1 is allowed to wait.

Deadlock Detection and Removal

The deadlock detection and removal approach runs a deadlock detection algorithm periodically and removes deadlock in case there is one. It does not check for deadlock when a transaction places a request for a lock. When a transaction requests a lock, the lock manager checks whether it is available. If it is available, the transaction is allowed to lock the data item; otherwise the transaction is allowed to wait.

Since there are no precautions while granting lock requests, some of the transactions may be deadlocked. To detect deadlocks, the lock manager periodically checks if the wait-forgraph has cycles. If the system is deadlocked, the lock manager chooses a victim transaction from each cycle. The victim is aborted and rolled back; and then restarted later. Some of the methods used for victim selection are −

- Choose the youngest transaction.
- Choose the transaction with fewest data items.
- Choose the transaction that has performed least number of updates.
- Choose the transaction having least restart overhead.

- Choose the transaction which is common to two or more cycles.

This approach is primarily suited for systems having transactions low and where fast response to lock requests is needed.

Deadlock Handling in Distributed Systems

Transaction processing in a distributed database system is also distributed, i.e. the same transaction may be processing at more than one site. The two main deadlock handling concerns in a distributed database system that are not present in a centralized system are **transaction location** and **transaction control**. Once these concerns are addressed, deadlocks are handled through any of deadlock prevention, deadlock avoidance or deadlock detection and removal.

Transaction Location

Transactions in a distributed database system are processed in multiple sites and use data items in multiple sites. The amount of data processing is not uniformly distributed among these sites. The time period of processing also varies. Thus the same transaction may be active at some sites and inactive at others. When two conflicting transactions are located in a site, it may happen that one of them is in inactive state. This condition does not arise in a centralized system. This concern is called transaction location issue.

This concern may be addressed by Daisy Chain model. In this model, a transaction carries certain details when it moves from one site to another. Some of the details are the list of tables required, the list of sites required, the list of visited tables and sites, the list of tables and sites that are yet to be visited and the list of acquired locks with types. After a transaction terminates by either commit or abort, the information should be sent to all the concerned sites.

Transaction Control

Transaction control is concerned with designating and controlling the sites required for processing a transaction in a distributed database system. There are many options regarding the choice of where to process the transaction and how to designate the center of control, like −

- One server may be selected as the center of control.
- The center of control may travel from one server to another.
- The responsibility of controlling may be shared by a number of servers.

Distributed Deadlock Prevention

Just like in centralized deadlock prevention, in distributed deadlock prevention approach, a transaction should acquire all the locks before starting to execute. This prevents deadlocks.

The site where the transaction enters is designated as the controlling site. The controlling site sends messages to the sites where the data items are located to lock the items. Then it waits for confirmation. When all the sites have confirmed that they have locked the data items, transaction starts. If any site or communication link fails, the transaction has to wait until they have been repaired.

Though the implementation is simple, this approach has some drawbacks −

- Pre-acquisition of locks requires a long time for communication delays. This increases the time required for transaction.
- In case of site or link failure, a transaction has to wait for a long time so that the sites recover. Meanwhile, in the running sites, the items are locked. This may prevent other transactions from executing.
- If the controlling site fails, it cannot communicate with the other sites. These sites continue to keep the locked data items in their locked state, thus resulting in blocking.

Distributed Deadlock Avoidance

As in centralized system, distributed deadlock avoidance handles deadlock prior to occurrence. Additionally, in distributed systems, transaction location and transaction control issues needs to be addressed. Due to the distributed nature of the transaction, the following conflicts may occur −

- Conflict between two transactions in the same site.
- Conflict between two transactions in different sites.

In case of conflict, one of the transactions may be aborted or allowed to wait as per distributed wait-die or distributed wound-wait algorithms.

Let us assume that there are two transactions, T1 and T2. T1 arrives at Site P and tries to lock a data item which is already locked by T2 at that site. Hence, there is a conflict at Site P. The algorithms are as follows −

- **Distributed Wound-Die**
    - If T1 is older than T2, T1 is allowed to wait. T1 can resume execution after Site P receives a message that T2 has either committed or aborted successfully at all sites.
    - If T1 is younger than T2, T1 is aborted. The concurrency control at Site P sends a message to all sites where T1 has visited to abort T1. The controlling site notifies the user when T1 has been successfully aborted in all the sites.
- **Distributed Wait-Wait**
    - If T1 is older than T2, T2 needs to be aborted. If T2 is active at Site P, Site P aborts and rolls back T2 and then broadcasts this message to other relevant sites. If T2 has left Site P but is active at Site Q, Site P broadcasts that T2 has been aborted; Site L then aborts and rolls back T2 and sends this message to all sites.
    - If T1 is younger than T1, T1 is allowed to wait. T1 can resume execution after Site P receives a message that T2 has completed processing.

Distributed Deadlock Detection

Just like centralized deadlock detection approach, deadlocks are allowed to occur and are removed if detected. The system does not perform any checks when a transaction places a lock request. For implementation, global wait-for-graphs are created. Existence of a cycle in the global wait-for-graph indicates deadlocks. However, it is difficult to spot deadlocks since transaction waits for resources across the network.

Alternatively, deadlock detection algorithms can use timers. Each transaction is associated with a timer which is set to a time period in which a transaction is expected to finish. If a transaction does not finish within this time period, the timer goes off, indicating a possible deadlock.

Another tool used for deadlock handling is a deadlock detector. In a centralized system, there is one deadlock detector. In a distributed system, there can be more than one deadlock detectors. A deadlock detector can find deadlocks for the sites under its control. There are three alternatives for deadlock detection in a distributed system, namely.

- **Centralized Deadlock Detector** − One site is designated as the central deadlock detector.
- **Hierarchical Deadlock Detector** − A number of deadlock detectors are arranged in hierarchy.
- **Distributed Deadlock Detector** − All the sites participate in detecting deadlocks and removing them.

Distributed Database Recovery

In order to recuperate from database failure, database management systems resort to a number of recovery management techniques. In this chapter, we will study the different approaches for database recovery.

The typical strategies for database recovery are −

- In case of soft failures that result in inconsistency of database, recovery strategy includes transaction undo or rollback. However, sometimes, transaction redo may also be adopted to recover to a consistent state of the transaction.
- In case of hard failures resulting in extensive damage to database, recovery strategies encompass restoring a past copy of the database from archival backup. A more current state of the database is obtained through redoing operations of committed transactions from transaction log.

Recovery from Power Failure

Power failure causes loss of information in the non-persistent memory. When power is restored, the operating system and the database management system restart. Recovery manager initiates recovery from the transaction logs.

In case of immediate update mode, the recovery manager takes the following actions −

- Transactions which are in active list and failed list are undone and written on the abort list.
- Transactions which are in before-commit list are redone.
- No action is taken for transactions in commit or abort lists.

In case of deferred update mode, the recovery manager takes the following actions −

- Transactions which are in the active list and failed list are written onto the abort list. No undo operations are required since the changes have not been written to the disk yet.
- Transactions which are in before-commit list are redone.
- No action is taken for transactions in commit or abort lists.

Recovery from Disk Failure

A disk failure or hard crash causes a total database loss. To recover from this hard crash, a new disk is prepared, then the operating system is restored, and finally the database is recovered using the database backup and transaction log. The recovery method is same for both immediate and deferred update modes.

The recovery manager takes the following actions −

- The transactions in the commit list and before-commit list are redone and written onto the commit list in the transaction log.
- The transactions in the active list and failed list are undone and written onto the abort list in the transaction log.

Checkpointing

Checkpoint is a point of time at which a record is written onto the database from the buffers. As a consequence, in case of a system crash, the recovery manager does not have to redo the transactions that have been committed before checkpoint. Periodical checkpointing shortens the recovery process.

The two types of checkpointing techniques are −

- Consistent checkpointing
- Fuzzy checkpointing

Consistent Checkpointing

Consistent checkpointing creates a consistent image of the database at checkpoint. During recovery, only those transactions which are on the right side of the last checkpoint are undone or

redone. The transactions to the left side of the last consistent checkpoint are already committed and needn't be processed again. The actions taken for checkpointing are −

- The active transactions are suspended temporarily.
- All changes in main-memory buffers are written onto the disk.
- A "checkpoint" record is written in the transaction log.
- The transaction log is written to the disk.
- The suspended transactions are resumed.

If in step 4, the transaction log is archived as well, then this checkpointing aids in recovery from disk failures and power failures, otherwise it aids recovery from only power failures.

Fuzzy Checkpointing

In fuzzy checkpointing, at the time of checkpoint, all the active transactions are written in the log. In case of power failure, the recovery manager processes only those transactions that were active during checkpoint and later. The transactions that have been committed before checkpoint are written to the disk and hence need not be redone.

Example of Checkpointing

Let us consider that in system the time of checkpointing is tcheck and the time of system crash is tfail. Let there be four transactions T_a, T_b, T_c and T_d such that −

- T_a commits before checkpoint.
- T_b starts before checkpoint and commits before system crash.
- T_c starts after checkpoint and commits before system crash.
- T_d starts after checkpoint and was active at the time of system crash.

The situation is depicted in the following diagram −

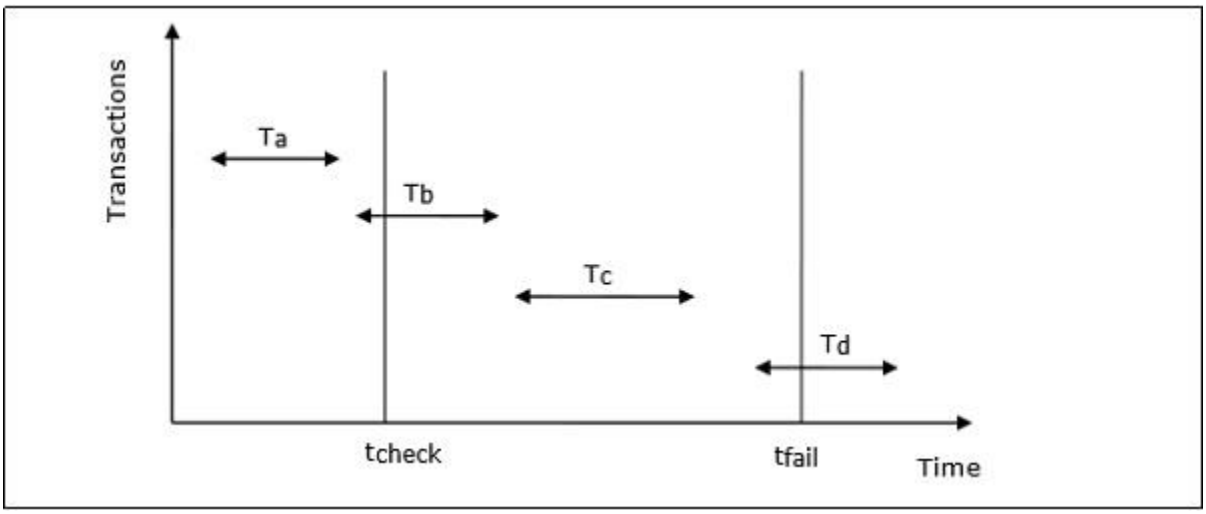

Fig 2.7 Checkpointing

The actions that are taken by the recovery manager are −

- Nothing is done with T_a.
- Transaction redo is performed for T_b and T_c.
- Transaction undo is performed for T_d.

Transaction Recovery Using UNDO / REDO

Transaction recovery is done to eliminate the adverse effects of faulty transactions rather than to recover from a failure. Faulty transactions include all transactions that have changed the database into undesired state and the transactions that have used values written by the faulty transactions.

Transaction recovery in these cases is a two-step process −

- UNDO all faulty transactions and transactions that may be affected by the faulty transactions.
- REDO all transactions that are not faulty but have been undone due to the faulty transactions.

Steps for the UNDO operation are −

- If the faulty transaction has done INSERT, the recovery manager deletes the data item(s) inserted.
- If the faulty transaction has done DELETE, the recovery manager inserts the deleted data item(s) from the log.

- If the faulty transaction has done UPDATE, the recovery manager eliminates the value by writing the before-update value from the log.

Steps for the REDO operation are −

- If the transaction has done INSERT, the recovery manager generates an insert from the log.
- If the transaction has done DELETE, the recovery manager generates a delete from the log.
- If the transaction has done UPDATE, the recovery manager generates an update from the log.

In a local database system, for committing a transaction, the transaction manager has to only convey the decision to commit to the recovery manager. However, in a distributed system, the transaction manager should convey the decision to commit to all the servers in the various sites where the transaction is being executed and uniformly enforce the decision. When processing is complete at each site, it reaches the partially committed transaction state and waits for all other transactions to reach their partially committed states. When it receives the message that all the sites are ready to commit, it starts to commit. In a distributed system, either all sites commit or none of them does.

The different distributed commit protocols are −

- One-phase commit
- Two-phase commit
- Three-phase commit

Distributed One-phase Commit

Distributed one-phase commit is the simplest commit protocol. Let us consider that there is a controlling site and a number of slave sites where the transaction is being executed. The steps in distributed commit are −

- After each slave has locally completed its transaction, it sends a “DONE” message to the controlling site.
- The slaves wait for “Commit” or “Abort” message from the controlling site. This waiting time is called **window of vulnerability**.

- When the controlling site receives “DONE” message from each slave, it makes a decision to commit or abort. This is called the commit point. Then, it sends this message to all the slaves.
- On receiving this message, a slave either commits or aborts and then sends an acknowledgement message to the controlling site.

Distributed Two-phase Commit

Distributed two-phase commit reduces the vulnerability of one-phase commit protocols. The steps performed in the two phases are as follows −

Phase 1: Prepare Phase

- After each slave has locally completed its transaction, it sends a “DONE” message to the controlling site. When the controlling site has received “DONE” message from all slaves, it sends a “Prepare” message to the slaves.
- The slaves vote on whether they still want to commit or not. If a slave wants to commit, it sends a “Ready” message.
- A slave that does not want to commit sends a “Not Ready” message. This may happen when the slave has conflicting concurrent transactions or there is a timeout.

Phase 2: Commit/Abort Phase

- After the controlling site has received “Ready” message from all the slaves −
    - The controlling site sends a “Global Commit” message to the slaves.
    - The slaves apply the transaction and send a “Commit ACK” message to the controlling site.
    - When the controlling site receives “Commit ACK” message from all the slaves, it considers the transaction as committed.
- After the controlling site has received the first “Not Ready” message from any slave −
    - The controlling site sends a “Global Abort” message to the slaves.

- The slaves abort the transaction and send a “Abort ACK” message to the controlling site.
- When the controlling site receives “Abort ACK” message from all the slaves, it considers the transaction as aborted.

Distributed Three-phase Commit

The steps in distributed three-phase commit are as follows −

Phase 1: Prepare Phase

The steps are same as in distributed two-phase commit.

Phase 2: Prepare to Commit Phase

- The controlling site issues an “Enter Prepared State” broadcast message.
- The slave sites vote “OK” in response.

Phase 3: Commit / Abort Phase

The steps are same as two-phase commit except that “Commit ACK”/”Abort ACK” message is not required.

Distributed Data Security

Database Security and Threats

Data security is an imperative aspect of any database system. It is of particular importance in distributed systems because of large number of users, fragmented and replicated data, multiple sites and distributed control.

Threats in a Database

- **Availability loss** − Availability loss refers to non-availability of database objects by legitimate users.
- **Integrity loss** − Integrity loss occurs when unacceptable operations are performed upon the database either accidentally or maliciously. This may happen while creating, inserting, updating or deleting data. It results in corrupted data leading to incorrect decisions.

- **Confidentiality loss** − Confidentiality loss occurs due to unauthorized or unintentional disclosure of confidential information. It may result in illegal actions, security threats and loss in public confidence.

Measures of Control

The measures of control can be broadly divided into the following categories −

- **Access Control** − Access control includes security mechanisms in a database management system to protect against unauthorized access. A user can gain access to the database after clearing the login process through only valid user accounts. Each user account is password protected.
- **Flow Control** − Distributed systems encompass a lot of data flow from one site to another and also within a site. Flow control prevents data from being transferred in such a way that it can be accessed by unauthorized agents. A flow policy lists out the channels through which information can flow. It also defines security classes for data as well as transactions.
- **Data Encryption** − Data encryption refers to coding data when sensitive data is to be communicated over public channels. Even if an unauthorized agent gains access of the data, he cannot understand it since it is in an incomprehensible format.

What is Cryptography?

Cryptography is the science of encoding information before sending via unreliable communication paths so that only an authorized receiver can decode and use it.

The coded message is called **cipher text** and the original message is called **plain text**. The process of converting plain text to cipher text by the sender is called encoding or **encryption**. The process of converting cipher text to plain text by the receiver is called decoding or **decryption**.

The entire procedure of communicating using cryptography can be illustrated through the following diagram −

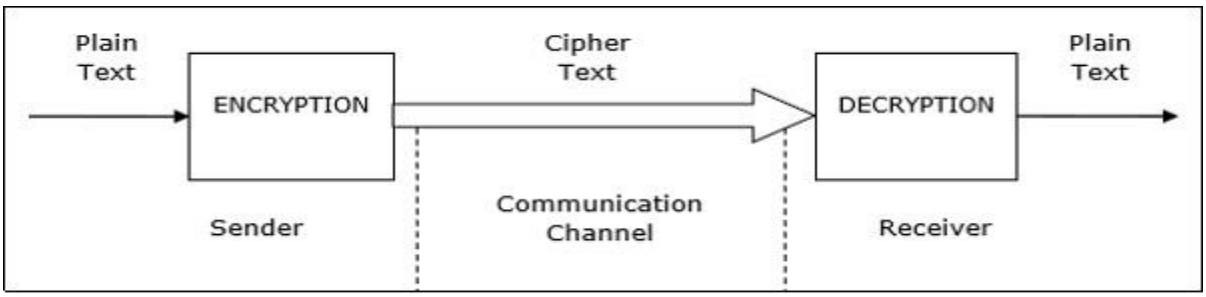

Fig 2.8 Cryptography

Conventional Encryption Methods

In conventional cryptography, the encryption and decryption is done using the same secret key. Here, the sender encrypts the message with an encryption algorithm using a copy of the secret key. The encrypted message is then send over public communication channels. On receiving the encrypted message, the receiver decrypts it with a corresponding decryption algorithm using the same secret key.

Security in conventional cryptography depends on two factors −

- A sound algorithm which is known to all.
- A randomly generated, preferably long secret key known only by the sender and the receiver.

The most famous conventional cryptography algorithm is **Data Encryption Standard** or **DES**.

The advantage of this method is its easy applicability. However, the greatest problem of conventional cryptography is sharing the secret key between the communicating parties. The ways to send the key are cumbersome and highly susceptible to eavesdropping.

Public Key Cryptography

In contrast to conventional cryptography, public key cryptography uses two different keys, referred to as public key and the private key. Each user generates the pair of public key and private key. The user then puts the public key in an accessible place. When a sender wants to sends a message, he encrypts it using the public key of the receiver. On receiving the encrypted

message, the receiver decrypts it using his private key. Since the private key is not known to anyone but the receiver, no other person who receives the message can decrypt it.

The most popular public key cryptography algorithms are **RSA** algorithm and **Diffie–Hellman** algorithm. This method is very secure to send private messages. However, the problem is, it involves a lot of computations and so proves to be inefficient for long messages.

The solution is to use a combination of conventional and public key cryptography. The secret key is encrypted using public key cryptography before sharing between the communicating parties. Then, the message is send using conventional cryptography with the aid of the shared secret key.

Digital Signatures

A Digital Signature (DS) is an authentication technique based on public key cryptography used in e-commerce applications. It associates a unique mark to an individual within the body of his message. This helps others to authenticate valid senders of messages.

Typically, a user's digital signature varies from message to message in order to provide security against counterfeiting. The method is as follows −

- The sender takes a message, calculates the message digest of the message and signs it digest with a private key.
- The sender then appends the signed digest along with the plaintext message.
- The message is sent over communication channel.
- The receiver removes the appended signed digest and verifies the digest using the corresponding public key.
- The receiver then takes the plaintext message and runs it through the same message digest algorithm.
- If the results of step 4 and step 5 match, then the receiver knows that the message has integrity and authentic.

A distributed system needs additional security measures than centralized system, since there are many users, diversified data, multiple sites and distributed control. In this chapter, we will look into the various facets of distributed database security.

In distributed communication systems, there are two types of intruders −

- **Passive eavesdroppers** − They monitor the messages and get hold of private information.
- **Active attackers** − They not only monitor the messages but also corrupt data by inserting new data or modifying existing data.

Security measures encompass security in communications, security in data and data auditing.

Communications Security

In a distributed database, a lot of data communication takes place owing to the diversified location of data, users and transactions. So, it demands secure communication between users and databases and between the different database environments.

Security in communication encompasses the following −

- Data should not be corrupt during transfer.
- The communication channel should be protected against both passive eavesdroppers and active attackers.
- In order to achieve the above stated requirements, well-defined security algorithms and protocols should be adopted.

Two popular, consistent technologies for achieving end-to-end secure communications are −

- Secure Socket Layer Protocol or Transport Layer Security Protocol.
- Virtual Private Networks (VPN).

Data Security

In distributed systems, it is imperative to adopt measure to secure data apart from communications. The data security measures are −

- **Authentication and authorization** − These are the access control measures adopted to ensure that only authentic users can use the database. To provide authentication digital certificates are used. Besides, login is restricted through username/password combination.
- **Data encryption** − The two approaches for data encryption in distributed systems are −

    - Internal to distributed database approach: The user applications encrypt the data and then store the encrypted data in the database. For using the stored data, the applications fetch the encrypted data from the database and then decrypt it.
    - External to distributed database: The distributed database system has its own encryption capabilities. The user applications store data and retrieve them without realizing that the data is stored in an encrypted form in the database.
- **Validated input** − In this security measure, the user application checks for each input before it can be used for updating the database. An un-validated input can cause a wide range of exploits like buffer overrun, command injection, cross-site scripting and corruption in data.

Data Auditing

A database security system needs to detect and monitor security violations, in order to ascertain the security measures it should adopt. It is often very difficult to detect breach of security at the time of occurrences. One method to identify security violations is to examine audit logs. Audit logs contain information such as −

- Date, time and site of failed access attempts.
- Details of successful access attempts.
- Vital modifications in the database system.
- Access of huge amounts of data, particularly from databases in multiple sites.

Web data management

The World Wide Web (“WWW” or “web” for short) has become a major repositoryof data and documents. the web represents a very large, dynamic, anddistributed data store and there are the obvious distributed data management issuesin accessing web data. The web, in its present form, can be viewed as two distinct yet related components.The first of these components is what is known as the *publicly indexable web*(PIW) that is composed of all static (and cross-linked) web pages that exist on webservers. These can be easily searched and indexed. The other component, whichis known as the *deep web* (or the *hidden web*), is composed of a huge number ofdatabases that encapsulate the data, hiding it from the outside world. The data in thehidden web are usually

retrieved by means of search interfaces where the user entersa query that is passed to the database server, and the results are returned to the useras a dynamically generated web pae. A portion of the deep web has come to beknown as the “dark web,” which consists of encrypted data and requires a particularbrowser such as Tor to access.

Web Graph Management

The web consists of “pages” that are connected by hyperlinks, and this structurecan be modeled as a directed graph that reflects the hyperlink structure. In thisgraph, commonly referred to as the *web graph*, static HTML web pages are thevertices and the links between the pages are represented as directed edges. Thecharacteristics of the web graph is important for studying data management issuessince the graph structure is exploited in web search, categorization and classification

of web content, and other web-related tasks. Theimportant characteristics of the web graph are the following:

(a) It is quite volatile.

(b) It is sparse. A graph is considered sparse if its average degree (i.e., the averageof the degrees of all of its vertices) is less than the number of vertices. This means that the each vertex of the graph has a limited number of neighbors, evenif the vertices are in general connected.

(c) It is “self-organizing.” The web contains a number of communities, eachof which consists of a set of pages that focus on a particular topic. Thesecommunities get organized on their own without any “centralized control,” andgive rise to the particular subgraphs in the web graph.

(d) It is a “small-world graph.” This property is related to sparseness—each nodein the graph may not have many neighbors (i.e., its degree may be small), butmany nodes are connected through intermediaries. Small-world networks werefirst identified in social sciences where it was noted that many people who arestrangers to each other are connected by intermediaries. This holds true in webgraphs as well in terms of the connectedness of the graph.

(e) It is a power law graph. The in- and out-degree distributions of the web graphfollow power law distributions. This means that the probability that a vertex hasin- (out-) degree $i$is proportional to $1/i\alpha$ for some $\alpha > 1$. The value of α isabout 2.1 for in-degree and about 7.2 for out-degree.

This brings us to a discussion of the structure of the web graph, which has a“bowtie” shape . It has a strongly connected component (the knot inthe middle) in which there is a path between each pair of pages.

Web Search

Web search involves finding “all” the web pages that are relevant (i.e., have contentrelated) to keyword(s) that a user specifies. Naturally, it is not possible to find all thepages, or even to know if one has retrieved all the pages; thus the search is performedon a database of web pages that have been collected and indexed. Since there areusually multiple pages that are relevant to a query, these pages are presented to the

user in ranked order of relevance as determined by the search engine. In every search engine the *crawler* plays one of the most crucial roles. A crawleris a program used by a search engine to scan the web on its behalf and collectdata about web pages. A crawler is given a starting set of pages—more accurately,it is given a set of Uniform Resource Locators (URLs) that identify these pages.

The crawler retrieves and parses the page corresponding to that URL, extracts any URLs in it, andadds these URLs to a queue. In the next cycle, the crawler extractsa URL from the queue (based on some order) and retrieves the corresponding page.This process is repeated until the crawler stops. A control module is responsible fordeciding which URLs should be visited next. The retrieved pages are stored in apage repository. The *indexer module* is responsible for constructing indexes on the pages that havebeen downloaded by the crawler.While many different indexes can be built, the twomost common ones are *text indexes* and *link indexes*. In order to construct a text

index, the indexer module constructs a large “lookup table” that can provide all theURLs that point to the pages where a given word occurs. A link index describes thelink structure of the web and provides information on the in-link and out-link stateof pages. The *ranking module* is responsible for sorting a large number of results so thatthose that are considered to be most relevant to the user’s search are presentedfirst. The problem of ranking has drawn increased interest in order to go beyondtraditional information retrieval (IR) techniques to address the special characteristicsof the web—web queries are usually small and they are executed over a vast amountof data.

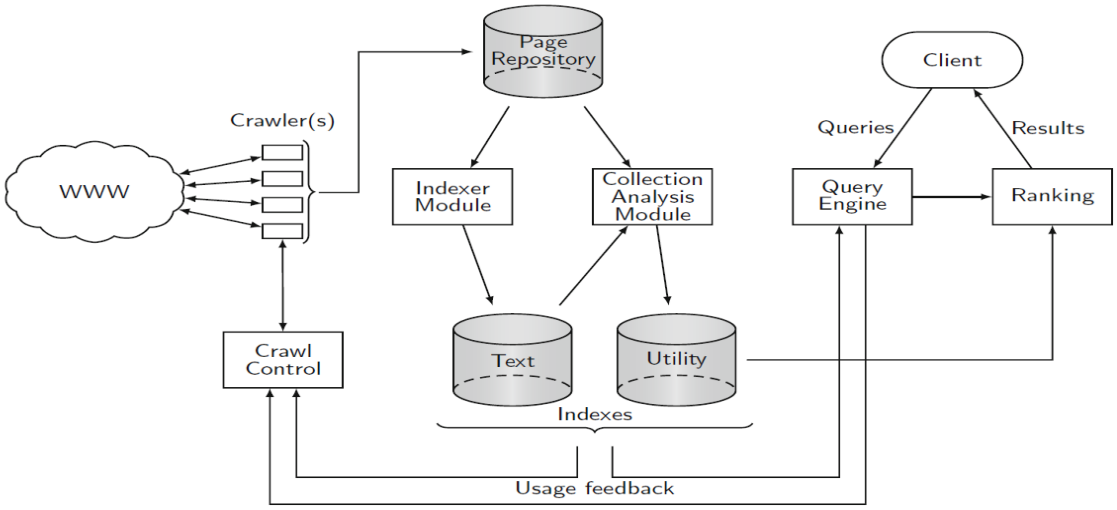

Fig 2.9 Web Data Management

Web Crawling

crawler scans the web on behalf of a search engine to extractinformation about the visited web pages. Given the size of the web, the changingnature of web pages, and the limited computing and storage capabilities of crawlers,it is impossible to crawl the entire web. Thus, a crawler must be designed to visit"most important" pages before others. The issue, then, is to visit the pages in someranked order of importance.There are a number of issues that need to be addressed in designing a crawler.Since the primary goal is to access more important pages before others, there needsto be some way of determining the importance of a page. This can be done bymeans of a measure that reflects the importance of a given page. These measurescan be static, such that the importance of a page is determined independent ofretrieval queries that will run against it, or dynamic in that they take the queries intoconsideration. Examples of static measures are those that determine the importanceof a page *Pi* with respect to the number of pages that point to *Pi* (referred to as*backlink*), or those that additionally take into account the importance of the backlinkpages as is done in the popular PageRank metric that is used by Google and others.A possible dynamicmeasure may be one that calculates the importance of a page*Pi* with respect its textual similarity to the query that is being evaluated using someof the well-known information retrieval similarity measures. Recall that thePageRank of a page *Pi* , denoted *PR(Pi)*, is simply the normalized sum of thePageRank of all *Pi* 's backlink pages (denoted as *BPi*) where the normalization foreach $Pj \in BPi$is over all of *Pj*'s forward links *FPj*:

$$PR(P_i) = \sum_{P_j \in B_{P_i}} \frac{PR(P_j)}{|F_{P_j}|}$$

Recall also that this formula calculates the rank of a page based on the backlinks,but normalizes the contribution of each backlinking page *Pj*using the number offorward links that *Pj*has. The idea here is that it is more important to be pointedat by pages conservatively link to other pages than by those who link to othersindiscriminately, but the "contribution" of a link from such a page needs to benormalized over all the pages that it points to.A second issue is how the crawler chooses the next page to visit once it hascrawled a particular page. As noted earlier, the crawler maintains a queue in whichit stores the URLs for the pages that it discovers as it analyzes each page. Thus,the issue is one of ordering the URLs in this queue. A number of strategies arepossible. One possibility is to visit the URLs in the order in which they werediscovered; this is referred to as the *breadth-first approach*. Another alternative isto use random ordering whereby the crawler chooses a URL randomly from amongthose that are in its queue of unvisited pages. Other alternatives are to use metricsthat combine ordering with importance ranking discussed above, such as backlinkcounts or PageRank.