DISTRIBUTED DATABASES (R22CSE3146)

III B.Tech I Semester (Information Technology)

UNIT – III

Transaction Management : Definition, properties of transaction, types of transactions, Distributed concurrency control: serializability, concurrency control mechanisms & algorithms, time - stamped & optimistic concurrency control Algorithms, deadlock Management.

Definition of a Transaction

- Transaction consists of a set of operations that performs a single logical unit of work in database environment.
- The *transaction* is a basic unit of consistent and reliable computing.
- A Transaction takes a database, performs an action on it and generate a new version of the database, causing state transition.
- This is similar to what a query does, except that if the database was consistent before the execution of the transaction, we can now guarantee that it will be consistent at the end of its execution regardless of the fact that

  (1) The transaction may have been executed concurrently with others, and
  (2) Failures may have occurred during its execution.
- A Transaction is considered to be made up of a sequence of read and write operations on the database together with computation steps.
- A Transaction may be thought of a program with embedded database access queries.

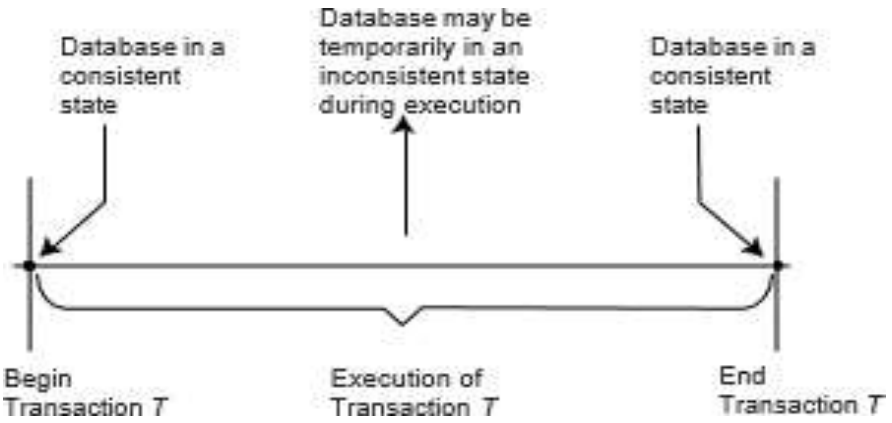


Fig : Transaction Model

Example : Airline Reservation System

FLIGHT relation that records the data about each flight

CUST relation for the customers who book flights

FC relation indicating which customers are on what flights.

- Let us also assume that the relation definitions are as follows (where the underlined attributes constitute the keys):
  FLIGHT(FNO, DATE, SRC, DEST, STSOLD, CAP)
  CUST(CNAME, ADDR, BAL)
  FC(FNO, DATE, CNAME, SPECIAL)
- FNO is the flight number, DATE denotes the flight date, SRC and DEST indicate the source and destination for the flight, STSOLD indicates the number of seats that have been sold on that flight, CAP denotes the passenger capacity on the flight, CNAME indicates the customer name whose address is stored in ADDR and whose account balance is in BAL, and SPECIAL corresponds to any special requests that the customer may have for a booking.
- Let us consider a simplified version of a typical reservation application, where a travel agent enters the flight number, the date, and a customer name, and asks for a reservation. The transaction to perform this function can be implemented as follows, where database accesses are specified in embedded SQL notation:

- **Begin transaction** Reservation

begin

- **input**(flight no, date, customer name); (1)
  EXEC SQL UPDATE FLIGHT (2)
  SET STSOLD = STSOLD + 1
  WHERE FNO = flight no AND DATE = date;
  EXEC SQL INSERT (3)
  INTO FC(FNO,DATE,CNAME,SPECIAL)
  VALUES (flight no, date, customer name, *null*);
  output("reservation completed") (4)

end.

end.

Explanation : First a point about notation. Even though we use embedded SQL, we do not follow its syntax very strictly. The lowercase terms are the program variables; the uppercase terms denote database relations and attributes as well as the SQL statements. Numeric constants are used as they are, whereas character constants are enclosed in quotes. Keywords of the host language are written in boldface, and *null* is a keyword for the null string.

Line (1) is to input the flight number, the date, and the customer name.

Line (2) updates the number of sold seats on the requested flight by one.

Line (3) inserts a tuple into the FC relation.

Here we assume that the customer is an old one, so it is not necessary to have an insertion into the CUST relation, creating a record for the client.

The keyword *null* in line (3) indicates that the customer has no special requests on this flight. Finally,

Line (4) reports the result of the transaction to the agent's terminal.

Characterization of Transactions

- Transactions read and write some data. This has been used as the basis for characterizing a transaction.
- The data items that a transaction reads are said to constitute its *read set* (*RS*).
- The data items that a transaction writes are said to constitute its *write set* (*WS*).
- The read set and write set of a transaction need not be mutually exclusive.
- The union of the read set and write set of a transaction constitutes its *base set* (*BS* = *RS* ∪ *WS*).

Example :

RS[Reservation] = {FLIGHT.STSOLD, FLIGHT.CAP}
WS[Reservation] = {FLIGHT.STSOLD, FC.FNO, FC.DATE,
FC.CNAME, FC.SPECIAL}
BS[Reservation] = {FLIGHT.STSOLD, FLIGHT.CAP,
FC.FNO, FC.DATE, FC.CNAME, FC.SPECIAL}

Formalization of the Transaction Concept

- The meaning of a transaction should be intuitively clear.
- To reason about transactions and about the correctness of the management algorithms, it is necessary to define the concept formally.

Example : Consider a simple transaction *T* that consists of the following steps:

Read(x)
Read(y)
$x \leftarrow x + y$
Write(x)
Commit

- One advantage of defining a transaction as a partial order is its correspondence to a **directed acyclic graph (DAG).**

- Thus a transaction can be specified as a DAG whose vertices are the operations of a transaction and whose arcs indicate the ordering relationship between a given pair of operations.

Example: The transaction can be represented as a DAG as depicted in Figure:

$$T = \{R(x), R(y), W(x), C\}$$

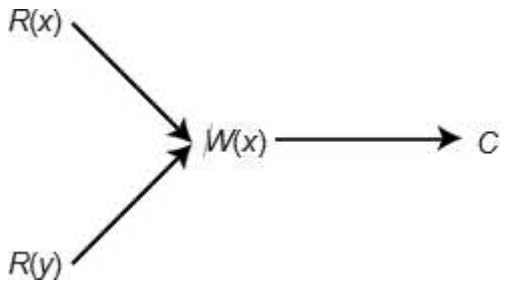
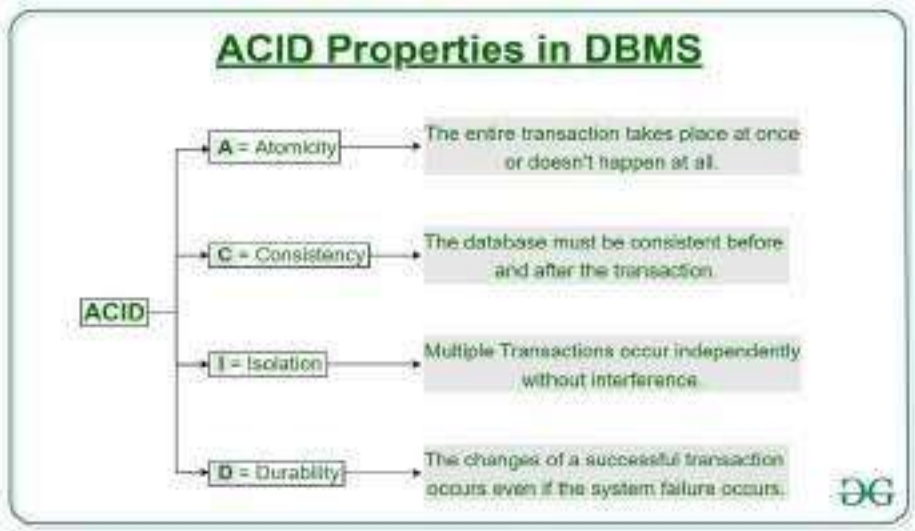


Fig: DAG Representation of a Transaction

Properties of Transactions

The consistency and reliability aspects of transactions are due to four properties: Together, these are commonly referred to as the ACID properties of transactions.

(1) atomicity,
(2) consistency,
(3) isolation, and
(4) durability.



1. **Atomicity**
   - *Atomicity* refers to the fact that a transaction is treated as a unit of operation.
   - Refers either all the transaction's actions are completed, or none of them are.
   - This is also known as the "**all-or-nothing property"**.
   - Atomicity requires that if the execution of a transaction is interrupted by any sort of failure, the DBMS will be responsible for determining what to do with the transaction upon recovery from the failure.
   - There are, of course, two possible courses of action:
     - it can either be terminated by completing the remaining actions,
     - or it can be terminated by undoing all the actions that have already been executed.
   - One can generally talk about two types of failures.
     - A transaction itself may fail due to input data errors, deadlocks, or other factors. In these cases either the transaction aborts itself, or the DBMS may abort it while handling deadlocks.
       Maintaining transaction atomicity in the presence of this type of failure is commonly called the ***transaction recovery***.
     - The second type of failure is caused by system crashes, such as media failures, processor failures, communication link breakages, power outages, and so on.
       Ensuring transaction atomicity in the presence of system crashes is called ***crash recovery.***
   - An important difference between the two types of failures is that during some types of system crashes, the information in volatile storage may be lost or inaccessible. Both types of recovery are parts of the reliability issue.

2. **Consistency**
   - Each user is responsible to ensure that their transaction would leave the database in a consistent state.
   - Transaction consistency is the responsibility of the user not the DBMS.
   - The *consistency* of a transaction is simply its correctness.
   - ATransaction is a correct program that maps one consistent database state to another.
   - This classification groups databases into four levels of consistency.
   - Then, based on the concept of dirty data, the four levels are defined as follows:

   - Degree 3: Transaction *T* sees *degree 3 consistency* if:
     1. *T* does not overwrite dirty data of other transactions.
     2. *T* does not commit any writes until it completes all its writes [i.e., until the end of transaction (EOT)].
     3. *T* does not read dirty data from other transactions.
     4. Other transactions do not dirty any data read by *T* before *T* completes.

- Degree 2: Transaction *T* sees *degree 2 consistency* if:
  1. *T* does not overwrite dirty data of other transactions.
  2. *T* does not commit any writes before EOT.
  3. *T* does not read dirty data from other transactions.

- Degree 1: Transaction *T* sees *degree 1 consistency* if:
  1. *T* does not overwrite dirty data of other transactions.
  2. *T* does not commit any writes before EOT.

- Degree 0: Transaction *T* sees *degree 0 consistency* if:
  1. *T* does not overwrite dirty data of other transactions."
- Of course, it is true that a higher degree of consistency encompasses all the lower degrees.
- The point in defining multiple levels of consistency is to provide application programmers the flexibility to define transactions that operate at different levels.
- Consequently, while some transactions operate at Degree 3 consistency level, others may operate at lower levels and may see, for example, dirty data.

3. Isolation

- *Isolation* is the property of transactions that requires each transaction to see a consistent database at all times.
- In other words, an executing transaction cannot reveal its results to other concurrent transactions before its commitment.
- There are a number of reasons for insisting on isolation.
- One has to do with maintaining the inter consistency of transactions.
- If two concurrent transactions access a data item that is being updated by one of them, it is not possible to guarantee that the second will read the correct value.

Example : Consider the following two concurrent transactions (T_1 and T_2), both of which access data item x. Assume that the value of x before they start executing is 50.

T_1: Read(x)	T_2: Read(x)
$x \leftarrow x+1$	$x \leftarrow x+1$
Write(x)	Write(x)
Commit	Commit

The following is one possible sequence of execution of the actions of these transactions:

T_1: Read(x)
T_1: $x \leftarrow x+1$

T_1: Write(x)
T_1: Commit
T_2: Read(x)
T_2: $x \leftarrow x+1$
T_2: Write(x)
T_2: Commit

In this case, there are no problems; transactions T_1 and T_2 are executed one after the other and transaction T_2 reads 51 as the value of x. Note that if, instead, T_2 executes before T_1, T_2 reads 51 as the value of x. So, if T_1 and T_2 are executed one after the other (regardless of the order), the second transaction will read 51 as the value of x and x will have 52 as its value at the end of execution of these two transactions. However, since transactions are executing concurrently,

The following execution sequence is also possible:

T_1: Read(x)
T_1: $x \leftarrow x+1$
T_2: Read(x)
T_1: Write(x)
T_2: $x \leftarrow x+1$
T_2: Write(x)
T_1: Commit
T_2: Commit

- In this case, transaction T_2 reads 50 as the value of x. This is incorrect since T_2 reads x while its value is being changed from 50 to 51. Furthermore, the value of x is 51 at the end of execution of T_1 and T_2 since T_2's Write will overwrite T_1's Write.
- Ensuring isolation by not permitting incomplete results to be seen by other transactions, as the previous example shows, solves the ***lost updates*** problem.
- This type of isolation has been called ***cursor stability***. In the example above, the second execution sequence resulted in the effects of T_1 being lost.
- A second reason for isolation is *cascading aborts*. If a transaction permits others to see its incomplete results before committing and then decides to abort, any transaction that has read its incomplete values will have to abort as well. This chain can easily grow and impose considerable overhead on the DBMS.
- As we move up the hierarchy of consistency levels, there is more isolation among transactions.
- Degree 0 provides very little isolation other than preventing lost updates. Transactions commit write operations before the entire transaction is completed (and committed), if an abort occurs after some writes are committed to disk, the updates to data items that have been committed will need to be undone. Since at this level other transactions are allowed to read the dirty data, it may be necessary to abort them as well.
- Degree 2 consistency avoids cascading aborts.

- Degree 3 provides full isolation which forces one of the conflicting transactions to wait until the other one terminates. Such execution sequences are called *strict.*
- ANSI, as part of the SQL2 (also known as SQL-92) standard specification, has defined a set of isolation levels. SQL isolation levels are defined on the basis of what ANSI call *phenomena* which are situations that can occur if proper isolation is not maintained. Three phenomena are specified:

Dirty Read:

- As defined earlier, dirty data refer to data items whose values have been modified by a transaction that has not yet committed.
- Consider the case where transaction T_1 modifies a data item value, which is then read by another transaction T_2 before T_1 performs a Commit or Abort.
- In case T_1 aborts, T_2 has read a value which never exists in the database.
- A precise specification of this phenomenon is as follows (where subscripts indicate the transaction identifiers)

$\ldots, W_1(x), \ldots, R_2(x), \ldots, C_1(\text{or } A_1), \ldots, C_2(\text{or } A_2)$

or

$\ldots, W_1(x), \ldots, R_2(x), \ldots, C_2(\text{or } A_2), \ldots, C_1(\text{or } A_1)$

Non-repeatable or Fuzzy read:

- Transaction T_1 reads a data item value. Another transaction T_2 then modifies or deletes that data item and commits.
- If T_1 then attempts to reread the data item, it either reads a different value or it can't find the data item at all; thus two reads within the same transaction T_1 return different results.
- A precise specification of this phenomenon is as follows:

$\ldots, R_1(x), \ldots, W_2(x), \ldots, C_1(\text{or } A_1), \ldots, C_2(\text{or } A_2)$

or

$\ldots, R_1(x), \ldots, W_2(x), \ldots, C_2(\text{or } A_2), \ldots, C_1(\text{or } A_1)$

Phantom:

- The phantom condition that was defined earlier occurs when T_1 does a search with a predicate and T_2 inserts new tuples that satisfy the predicate.
- Again, the precise specification of this phenomenon is (where P is the search predicate)

$\ldots, R_1(P), \ldots, W_2(y \text{ in } P), \ldots, C_1(\text{or } A_1), \ldots, C_2(\text{or } A_2)$

or

$\ldots, R_1(P), \ldots, W_2(y \text{ in } P), \ldots, C_2(\text{ or } A_2), \ldots, C_1(\text{or } A_1)$

- Based on these phenomena, the isolation levels are defined as follows. The objective of defining multiple isolation levels is the same as defining multiple consistency levels.
- **Read uncommitted :** For transactions operating at this level all three phenomena are possible.
- **Read committed :** Fuzzy reads and phantoms are possible, but dirty reads are not.
- **Repeatable read :** Only phantoms are possible.
- **Anomaly serializable :** None of the phenomena are possible.
- ANSI SQL standard uses the term "serializable" rather than "anomaly serializable."
- One non-serializable isolation level that is commonly implemented in commercial products is *snapshot isolation* Snapshot isolation provides repeatable reads, but not serializable isolation.
- Each transaction "sees" a snapshot of the database when it starts and its reads and writes are performed on this snapshot – thus the writes are not visible to other transactions and it does not see the writes of other transactions.

4. Durability

- *Durability* refers to that property of transactions which ensures that once a transaction commits, its results are permanent and cannot be erased from the database.
- Therefore, the DBMS ensures that the results of a transaction will survive subsequent system failures.
- The durability property brings forth the issue of ***database recovery***, that is, how to recover the database to a consistent state where all the committed actions are reflected.

Types of Transactions

Transactions have been classified according to a number of criteria.

1. First one is the duration of transactions (Timing).
2. Second is with respect to the organization of the read and write actions
3. Third one is according to their structure.

First one criterion is the duration of transactions

- Accordingly, transactions may be classified as *online* or *batch*. These two classes are also called *short-life* and *long-life* transactions.

- **Online transactions** are characterized by very short execution/response times (typically, on the order of a couple of seconds) and by access to a relatively small portion of the database.
- This class of transactions probably covers a large majority of current transaction applications.
- Examples : Banking transactions and Airline reservation transactions.
- **Batch transactions** take longer to execute (response time being measured in minutes, hours, or even days) and access a larger portion of the database.
- Typical applications that might require batch transactions are design databases.
- Examples : statistical applications, report generation, complex queries, and image processing. Along this dimension, one can also define a *conversational* transaction, which is executed by interacting with the user issuing it.

Second is with respect to the organization of the read and write actions

- Another classification that has been proposed is with respect to the organization of the read and write actions.
- Examples : considered so far intermix their read and write actions without any specific ordering. We call this type of transactions *general*.
- If the transactions are restricted so that all the read actions are performed before any write action, the transaction is called a *two-step* transaction.
- Similarly, if the transaction is restricted so that a data item has to be read before it can be updated (written), the corresponding class is called *restricted* (or *read-before-write*). If a transaction is both two- step and restricted, it is called a *restricted two-step* transaction.

Third one is according to their structure.

- Finally, there is the *action* model of transactions, which consists of the restricted class with the further restriction that each ⟨read, write⟩ pair be executed atomically.
- ***Example:*** The following are some examples of the above-mentioned models. We omit the declaration and commit commands.

General:

$T_1 : \{R(x), R(y), W(y), R(z), W(x), W(z), W(w), C\}$

Two-step:

$T_2 : \{R(x), R(y), R(z), W(x), W(z), W(y), W(w), C\}$

Restricted:

$T_3 : \{R(x), R(y), W(y), R(z), W(x), W(z), R(w), W(w), C\}$

Note that T_3 has to read w before writing.

Two-step restricted:

$T_4 : \{R(x), R(y), R(z), R(w), W(x), W(z), W(y), W(w), C\}$

Action:

$T_5 : \{[R(x), W(x)], [R(y), W(y)], [R(z), W(z)], [R(w), W(w)], C\}$

Note that each pair of actions within square brackets is executed atomically.

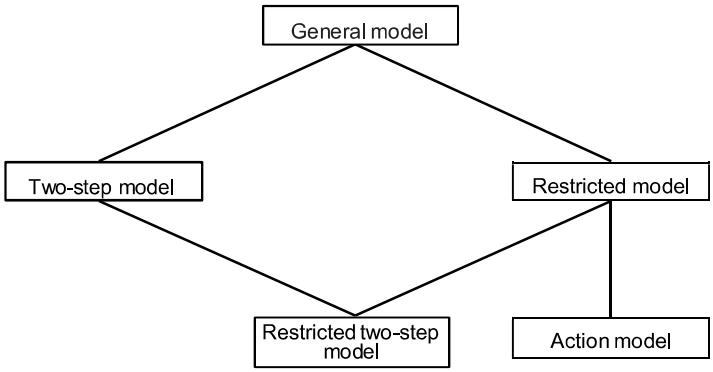


Fig : Various transaction Models

Transactions can also be classified according to their structure. We distinguish three broad categories in increasing complexity:

1. *Flat transactions*,
2. *Nested transactions,*
3. *Workflow models.*

Flat Transactions

- A client makes requests to multiple servers in a flat transaction
- Flat transactions have a single start point (**Begin transaction**) and a single termination point (**End transaction**).
- Transaction T, for example, is a flat transaction that performs operations on objects in servers X, Y, and Z.
  Before moving on to the next request, a flat client transaction completes the previous one.
- As a result, each transaction visits the server object in order. A transaction can only wait for one object at a time when servers utilize locking.
- They are usually very simple and are generally used for short activities rather than larger ones.

- It is beneficial to commit partial results. Bulk updates can be expensive to undo all updates.

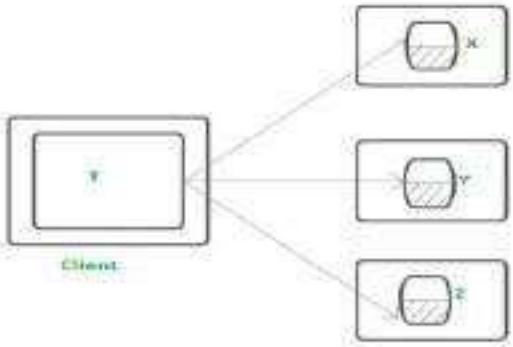


Fig : Flat Transaction

Nested Transactions

- In this Transaction model is to permit a transaction to include other transactions with their own begin and commit points. Such transactions are called *nested* transactions.
- These transactions that are embedded in another one are usually called *subtransactions*.
- *Example :* Let us extend the reservation transaction. Most travel agents will make reservations for hotels and car rentals in addition to the flights. If one chooses to specify all of this as one transaction, the reservation transaction would have the following structure:

Begin transaction Reservation
begin
Begin transaction Airline
...
end. {Airline}
Begin transaction Hotel
...
end. {Hotel}
Begin transaction Car
...
end. {Car}
end.
end.

- The level of nesting is generally open, allowing subtransactions themselves to have nested transactions. This generality is necessary to support application areas where transactions are more complex than in traditional data processing.
- Based on the termination characteristics Nested Transaction is classified into :

- ✓ *Closed Nested Transaction*
- ✓ *Open Nested Transaction*

Closed nested transactions : Commit in a bottom-up fashion through the root. Thus, a nested subtransaction begins *after* its parent and finishes *before* it, and the commitment of the subtransactions is conditional upon the commitment of the parent. The semantics of these transactions enforce atomicity at the top-most level.

Open nesting transactions : relaxes the top-level atomicity restriction of closed nested transactions. Therefore, an open nested transaction allows its partial results to be observed outside the transaction. Sagas and split transactions are examples of open nesting. A saga is a "sequence of transactions that can be interleaved with other transactions".

The advantages of nested transactions are the following.

- First, they provide a higher-level of concurrency among transactions. Since a transaction consists of a number of other transactions, more concurrency is possible within a single transaction.
- A second argument in favor of nested transactions is related to recovery. It is possible to recover independently from failures of each subtransaction. This limits the damage to a smaller part of the transaction, making it less costly to recover.
- Finally, it is possible to create new transactions from existing ones simply by inserting the old one inside the new one as a subtransaction.

Workflows

- Flat transactions model relatively simple and short activities very well. However, they are less appropriate for modeling longer and more elaborate activities. That is the reason for the development of the various nested transaction models.
- These extensions are not sufficiently powerful to model business activities: "after several decades of data processing, we have learned that we have not won the battle of modeling and automating complex enterprises".
- To meet these needs, more complex transaction models which are combinations of open and nested transactions have been proposed. There are well-justified arguments for not calling these transactions, since they hardly follow any of the ACID properties; a more appropriate name that has been proposed is a *workflow.*
- A working definition is that a workflow is "a collection of *tasks* organized to accomplish some business process." Three types of workflows are identified:

❖ ***Human-oriented workflows***, which involve humans in performing the tasks. The system support is provided to facilitate collaboration and coordination among humans, but it is the humans themselves who are ultimately responsible for the consistency of the actions.

- ***System-oriented workflows*** are those that consist of computation-intensive and specialized tasks that can be executed by a computer. The system support in this case is substantial and involves concurrency control and recovery, automatic task execution, notification, etc.
- ***Transactional workflows*** range in between human-oriented and system- oriented workflows and borrow characteristics from both. They involve "coordinated execution of multiple tasks that
  - may involve humans,
  - require access to HAD [heterogeneous, autonomous, and/or distributed] systems, and
  - support selective use of transactional properties [i.e., ACID properties] for individual tasks or entire workflows."
- Among the features of transactional workflows, the selective use of transactional properties is particularly important as it characterizes possible relaxations of ACID properties.

Example: Let us further extend the reservation transaction of Example 10.3. The entire reservation activity consists of the following taks and involves the follow- ing data:

- Customer request is obtained (task T_1) and Customer Database is accessed to obtain customer information, preferences, etc.;
- Airline reservation is performed (T_2) by accessing the Flight Database;
- Hotel reservation is performed (T_3), which may involve sending a message to the hotel involved;
- Auto reservation is performed (T_4), which may also involve communication with the car rental company;
- Bill is generated (T_5) and the billing info is recorded in the billing database.
- Figure depicts this workflow where there is a serial dependency of T_2 on T_1, and T_3, T_4 on T_2; however, T_3 and T_4 (hotel and car reservations) are performed in parallel and T_5 waits until their completion.

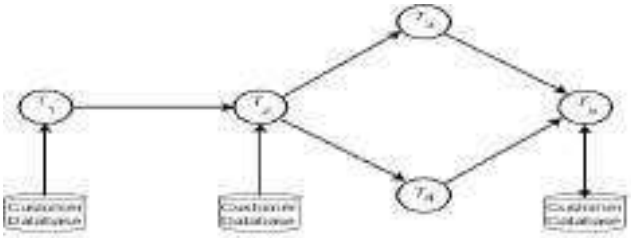
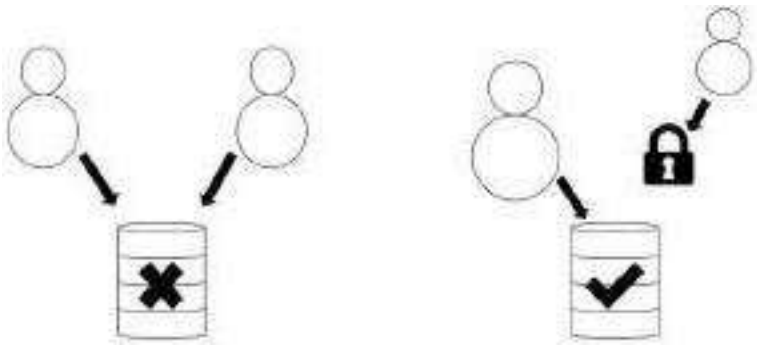


Fig : Workflow

Distributed Concurrency Control

- A Situation in which two or more persons access the same records simultaneously is called concurrency.
- Concurrency control involve the synchronization of access to the distributed database, such that the integrity of the database is maintained.
- Concurrency control deals with the isolation and consistency properties of transactions.
- The distributed concurrency control mechanism of a distributed DBMS ensures that the consistency of the database and it is maintained in a multiuser distributed environment.
- The level of concurrency (i.e., the number of concurrent transactions) is probably the most important parameter in distributed systems.
- The concurrency control mechanism attempts to find a suitable trade-off between maintaining the consistency of the database and maintaining a high level of concurrency.
- Serializability is the most widely accepted correctness criterion for concurrency control algorithms.



Serializability Theory

- The primary function of a concurrency controller is to generate a serializable history for the execution of pending transactions.
- To devise algorithms that are guaranteed to generate only serializable histories.
- Serializability theory extends in a straightforward manner to the non-replicated (or partitioned) distributed databases.
- The history of transaction execution at each site is called a *local history*.
- If the database is not replicated and each local history is serializable, their union (called the *global history*) is also serializable as long as local serialization orders are identical.

Example : We will give a very simple example to demonstrate the point. Consider two bank accounts, x (stored at Site 1) and y (stored at Site 2), and the following two

transactions where T_1 transfers \$100 from x to y, while T_2 simply reads the balances of x and y:

T_1:	T_2:
Read(x)	Read(x)
$x \leftarrow x - 100$	Read(y)
Write(x)	Commit
Read(y)	
$y \leftarrow y + 100$	
Write(y)	
Commit	

Obviously, both of these transactions need to run at both sites. Consider the following two histories that may be generated locally at the two sites (H_i is the history at Site i):

$H_1 = \{R_1(x), W_1(x), R_2(x)\}$
$H_2 = \{R_1(y), W_1(y), R_2(y)\}$

- Both of these histories are serializable; indeed, they are serial. Therefore, each represents a correct execution order.
- The serialization order for both are the same $T_1 \rightarrow T_2$. Therefore, the global history that is obtained is also serializable with the serialization order $T_1 \rightarrow T_2$.
- However, if the histories generated at the two sites are as follows, there is a problem:

$H1 = \{R_1(x), W_1(x), R_2(x)\}$
$H2 = \{R_2(y), R_1(y), W_1(y)\}$

- Although each local history is still serializable, the serialization orders are different: H' serializes T_1 before T_2 while H' serializes T_2 before T_1. Therefore, there can be no global history that is serializable.
- A weaker version of serializability that has gained importance in recent years is *snapshot isolation* that is now provided as a standard consistency criterion in a number of commercial systems.
- Snapshot isolation allows read transactions (queries) to read stale data by allowing them to read a snapshot of the database that reflects the committed data at the time the read transaction starts.
- Consequently, the reads are never blocked by writes, even though they may read old data that may be dirtied by other transactions that were still running when the snapshot was taken.
- Hence, the resulting histories are not serializable, but this is accepted as a reasonable tradeoff between a lower level of isolation and better performance.

Concurrency Control Mechanisms & Algorithms

- There are a number of ways that the concurrency control approaches can be classified.
- One obvious classification criterion is the mode of database distribution.
- Some algorithms that have been proposed require a fully replicated database, while others can operate on partially replicated or partitioned databases.
- The concurrency control algorithms may also be classified according to network topology, such as those requiring a communication subnet with broadcasting capability or those working in a star-type network or a circularly connected network.
- The corresponding breakdown of the concurrency control algorithms results in two classes:
  - those algorithms that are based on mutually exclusive access to shared data (locking),
  - those that attempt to order the execution of the transactions according to a set of rules (protocols).
- However, these primitives may be used in algorithms with two different viewpoints: the pessimistic view that many transactions will conflict with each other, or the optimistic view that not too many transactions will conflict with one another.
- Group the concurrency control mechanisms into two broad classes:
  - Pessimistic concurrency control methods
  - Optimistic concurrency control methods.
- ***Pessimistic* algorithms** synchronize the concurrent execution of transactions early in their execution life cycle.
- ***Optimistic* algorithms** delay the synchronization of transactions until their termination.
- The pessimistic group consists of *locking-based* algorithms, *ordering* (or *transaction ordering*) *based* algorithms, and *hybrid* algorithms.
- The optimistic group can, similarly, be classified as locking-based or timestamp ordering-based. This classification is depicted in Figure

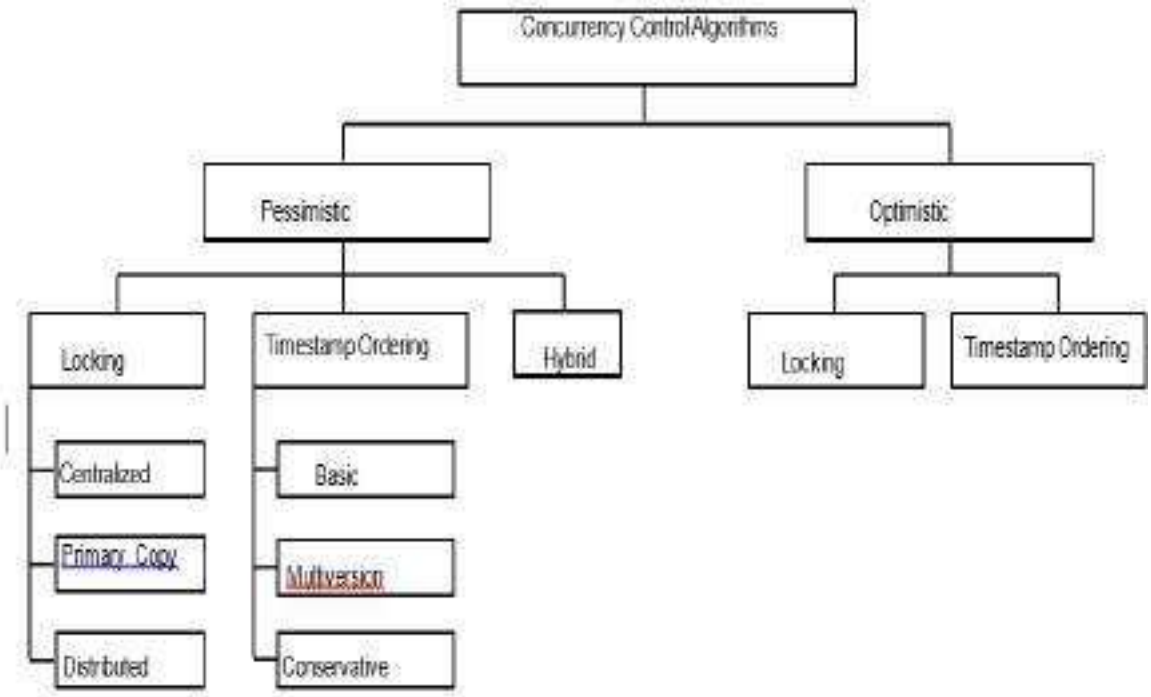


Fig : Classification of Concurrency Control Algorithms

- In the *locking-based* approach, the synchronization of transactions is achieved by employing physical or logical locks on some portion or granule of the database.
- This class is subdivided further according to where the lock management activities are performed: *centralized* and *decentralized* (or *distributed*) *locking*.
- The *timestamp ordering* (TO) class involves organizing the execution order of transactions so that they maintain transaction consistency.
- This ordering is maintained by assigning timestamps to both the transactions and the data items that are stored in the database. These algorithms can be *basic TO*, *multiversion TO*, or *conservative TO*.
- We should indicate that in some locking-based algorithms, timestamps are also used. This is done primarily to improve efficiency and the level of concurrency. We call these *hybrid* algorithms.

Locking-Based Concurrency Control Algorithms

- The main idea of locking-based concurrency control is to ensure that a data item that is shared by conflicting operations is accessed by one operation at a time. This is accomplished by associating a "lock" with each lock unit.
- This lock is set by a transaction before it is accessed and is reset at the end of its use.

- Obviously a lock unit cannot be accessed by an operation if it is already locked by another.
- There are two types of locks (commonly called *lock modes*) associated with each lock unit:
  - ✓ *read lock* (*rl*)
  - ✓ *write lock* (*wl*).
- A transaction T_i that wants to read a data item contained in lock unit x obtains a read lock on x [denoted $rl_i(x)$].
- The same happens for write operations.
- Two lock modes are *compatible* if two transactions that access the same data item can obtain these locks on that data item at the same time. read locks are compatible, whereas read-write or write-write locks are not. Therefore, it is possible, for example, for two transactions to read the same data item concurrently.

	$rl_i(x)$	$wl_i(x)$
$rl_j(x)$	compatible	not compatible
$wl_j(x)$	not compatible	not compatible

Fig : Compatibility Matrix of Lock Modes

- The distributed DBMS not only manages locks but also handles the lock management responsibilities on behalf of the transactions.
- It means users do not need to specify when a data item needs to be locked; the distributed DBMS takes care of that every time the transaction issues a read or write operation.

Example : Consider the following two transactions:

T_1: Read(x)	T_2: Read(x)
$x \leftarrow x + 1$	$x \leftarrow x * 2$
Write(x)	Write(x)
Read(y)	Read(y)
$y \leftarrow y - 1$	$y \leftarrow y * 2$
Write(y)	Write(y)
Commit	Commit

- The following is a valid history that a lock manager employing the locking algorithm may generate:

$$H = \{wl_1(x), R_1(x), W_1(x), lr_1(x), wl_2(x), R_2(x), w_2(x), lr_2(x), wl_2(y),$$

$$R_2(y), W_2(y), lr_2(y), wl_1(y), R_1(y), W_1(y), lr_1(y)\}$$

- where $lr_i(z)$ indicates the release of the lock on z that transaction T_i holds.
- The problem with history H is the locking algorithm releases the locks that are held by a transaction (say, T_i) as soon as the associated database command (read or write) is executed, and that lock unit (say x) no longer needs to be accessed.
- However, the transaction itself is locking other items (say, y), after it releases its lock on x. Even though this may seem to be advantageous from the viewpoint of increased concurrency, it permits transactions to interfere with one another, resulting in the loss of isolation and atomicity.
- Hence, we go for *two-phase locking* (2PL).
- The two-phase locking rule simply states that no transaction should request a lock after it releases one of its locks.
- 2PL algorithms execute transactions in two phases. Each transaction has a *growing phase*, where it obtains locks and accesses data items, and a *shrinking phase*, during which it releases locks.
- The *lock point* is the moment when the transaction has achieved all its locks but has not yet started to release any of them.
- Thus the lock point determines the end of the growing phase and the beginning of the shrinking phase of a transaction.
- It has been proven that any history generated by a concurrency control algorithm that obeys the 2PL rule is serializable.

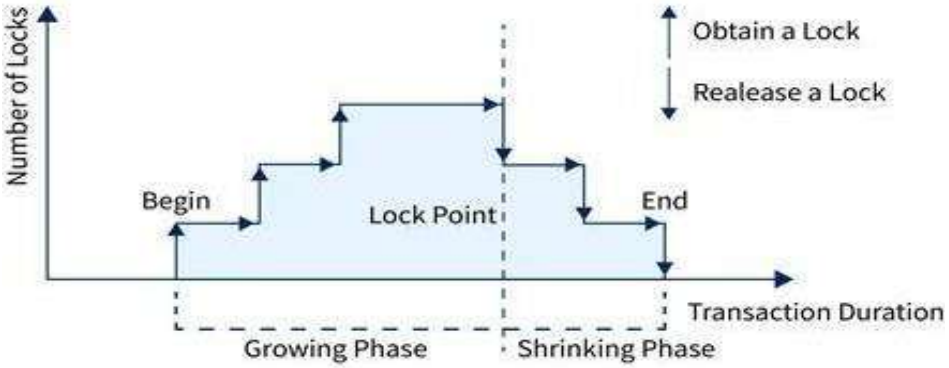


Fig 1 : 2PL Lock Graph

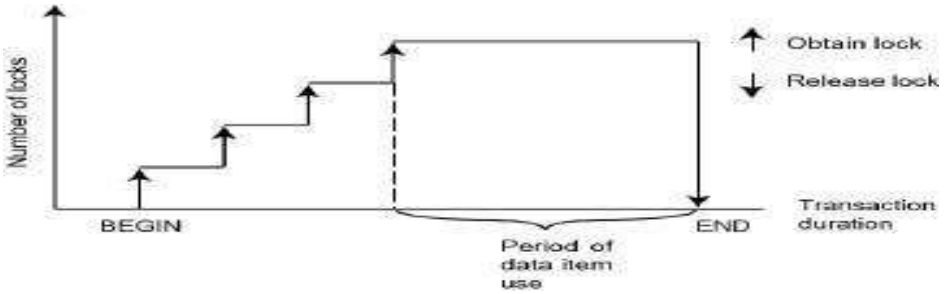


Fig 2: Strict 2PL Lock Graph.

- Figure 1 : it indicates that the lock manager releases locks as soon as access to that data item has been completed. This permits other transactions awaiting access to go ahead and lock it, thereby increasing the degree of concurrency.
- Figure 2 : if the transaction aborts after it releases a lock, it may cause other transactions that may have accessed the unlocked data item to abort as well. This is known as *cascading aborts*. These problems may be overcome by *strict two-phase locking*, which releases all the locks together when the transaction terminates (commits or aborts).

Centralized 2PL

- One way of doing this is to delegate lock management responsibility to a single site only. This means that only one of the sites has a lock manager; the transaction managers at the other sites communicate with it rather than with their own lock managers. This approach is also known as the *primary site* 2PL algorithm
- The communication between the cooperating sites in executing a transaction according to a centralized 2PL (C2PL) algorithm.
- This communication is between the transaction manager at the site where the transaction is initiated (called the *coordinating* TM), the lock manager at the central site, and the data processors (DP) at the other participating sites.
- The participating sites are those that store the data item and at which the operation is to be carried out. The order of messages is denoted in the figure.

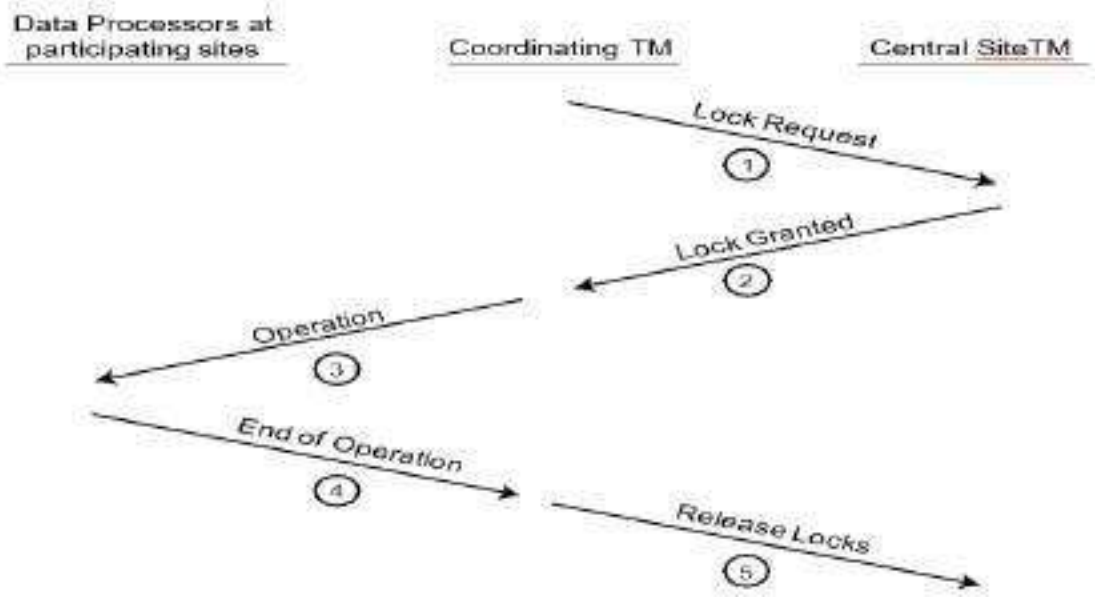


Fig : Communication Structure of Distributed 2PL

Distributed 2PL

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

Timestamp-Based Concurrency Control Algorithms

- To establish timestamp ordering, the transaction manager assigns each transaction T_i a unique timestamp, $ts(T_i)$, at its initiation.
- A timestamp is a simple identifier that serves to identify each transaction uniquely and is used for ordering.
- Uniqueness is only one of the properties of timestamp generation.
- The second property is monotonicity. Two timestamps generated by the same transaction manager should be monotonically increasing.
- There are a number of ways that timestamps can be assigned. To maintain uniqueness, each site appends its own identifier to the counter value. Thus the timestamp is a two-tuple of the form ⟨local counter value, site identifier⟩.
- It is simple to order the execution of the transactions' operations according to their timestamps. Formally, the timestamp ordering (TO) rule can be specified as follows:
- **TO Rule.** Given two conflicting operations O_{ij} and O_{kl} belonging, respectively, to transactions T_i and T_k, O_{ij} is executed before O_{kl} if and only if $ts(T_i) < ts(T_k)$. In this case T_i is said to be the *older* transaction and T_k is said to be the *younger* one.
- A scheduler that enforces the TO rule checks each new operation against conflicting operations that have already been scheduled. If the new operation belongs 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 entire transaction to restart with a *new* timestamp.
- A timestamp ordering scheduler that operates in this fashion is guaranteed to generate serializable histories. However, this comparison between the transaction timestamps can be performed only if the scheduler has received all the operations

to be scheduled. If operations come to the scheduler one at a time (which is the realistic case), it is necessary to be able to detect, in an efficient manner, if an operation has arrived out of sequence. To facilitate this check, each data item x is assigned two timestamps:

- a *read timestamp* [$rts(x)$], which is the largest of the timestamps of the transactions that have read x, and
- a *write timestamp* [$wts(x)$], which is the largest of the timestamps of the transactions that have written (updated) x. It is now sufficient to compare the timestamp of an operation with the read and write timestamps of the data item that it wants to access to determine if any transaction with a larger timestamp has already accessed the same data item.
- The transaction manager is responsible for assigning a timestamp to each new transaction and attaching this timestamp to each database operation that it passes on to the scheduler.
- The latter component is responsible for keeping track of read and write timestamps as well as performing the serializability check.

Basic TO Algorithm

- The basic TO algorithm is a straightforward implementation of the TO rule.
- The coordinating transaction manager assigns the timestamp to each transaction, determines the sites where each data item is stored, and sends the relevant operations to these sites.
- The histories at each site simply enforce the TO rule.
- A transaction one of whose operations is rejected by a scheduler is restarted by the transaction manager with a new timestamp. This ensures that the transaction has a chance to execute in its next try.
- Since the transactions never wait while they hold access rights to data items, the basic TO algorithm never causes deadlocks.
- The penalty of deadlock freedom is potential restart of a transaction numerous times.
- There is an alternative to the basic TO algorithm that reduces the number of restarts.
- When an accepted operation is passed on to the data processor, the scheduler needs to refrain from sending another conflicting, but acceptable operation to the data processor until the first is processed and acknowledged.
- This is a requirement to ensure that the data processor executes the operations in the order in which the scheduler passes them on. Otherwise, the read and write timestamp values for the accessed data item would not be accurate.
- ***Example*** ∴ Assume that the TO scheduler first receives $W_i(x)$ and then receives $W_j(x)$, where $ts(T_i) < ts(T_j)$. The scheduler would accept both operations and pass them on to the data processor. The result of these two operations is that $wts(x) = ts(T_j)$, and we then expect the effect of $W_j(x)$ to be represented in the database. However, if the data processor does not execute them in that order, the effects on

the database will be wrong.

- The scheduler can enforce the ordering by maintaining a queue for each data item that is used to delay the transfer of the accepted operation until an acknowledgment is received from the data processor regarding the previous operation on the same data item.
- Such a complication does not arise in 2PL-based algorithms because the lock manager effectively orders the operations by releasing the locks only after the operation is executed. In one sense the queue that the TO scheduler maintains may be thought of as a lock. However, this does not imply that the history generated by a TO scheduler and a 2PL scheduler would always be equivalent. There are some histories that a TO scheduler would generate that would not be admissible by a 2PL history.
- Remember that in the case of strict 2PL algorithms, the releasing of locks is delayed further, until the commit or abort of a transaction. It is possible to develop a strict TO algorithm by using a similar scheme. For example, if $W_i(x)$ is accepted and released to the data processor, the scheduler delays all $R_j(x)$ and $W_j(x)$ operations
- (for all T_j) until T_i terminates (commits or aborts).

Conservative TO Algorithm

- We indicated in the preceding section that the basic TO algorithm never causes operations to wait, but instead, restarts them. We also pointed out that even though this is an advantage due to deadlock freedom, it is also a disadvantage, because numerous restarts would have adverse performance implications.
- The conservative TO algorithms attempt to lower this system overhead by reducing the number of transaction restarts.
- Let us first present a technique that is commonly used to reduce the probability of restarts.
- Remember that a TO scheduler restarts a transaction if a younger conflicting transaction is already scheduled or has been executed.
- **Example :** one site is comparatively inactive relative to the others and does not issue transactions for an extended period. In this case its timestamp counter indicates a value that is considerably smaller than the counters of other sites.
- If the TM at this site then receives a transaction, the operations that are sent to the histories at the other sites will almost certainly be rejected, causing the transaction to restart.
- Furthermore, the same transaction will restart repeatedly until the timestamp counter value at its originating site reaches a level of parity with the counters of other sites.
- Each transaction manager can send its remote operations, rather than histories, to the transaction managers at the other sites.
- The receiving transaction managers can then compare their own counter values

with that of the incoming operation. Any manager whose counter value is smaller than the incoming one adjusts its own counter to one more than the incoming one.

- if system clocks are used instead of counters, this approximate synchronization may be achieved automatically as long as the clocks are of comparable speeds.
- We will consider one possible implementation of the conservative TO algorithm ;
- Assume that each scheduler maintains one queue for each transaction manager in the system. The scheduler at site *i* stores all the operations that it receives from the transaction manager at site *j* in queue Q_{ij}.
- Scheduler *i* has one such queue for each *j*. When an operation is received from a transaction manager, it is placed in its appropriate queue in increasing timestamp order.
- The histories at each site execute the operations from these queues in increasing timestamp order.
- This scheme will reduce the number of restarts, but it will not guarantee that they will be eliminated completely. Consider the case where at site *i* the queue for site *j* (Q_{ij}) is empty. The scheduler at site *i* will choose an operation [say, $R(x)$] with the smallest timestamp and pass it on to the data processor.
- However, site *j* may have sent to *i* an operation [say, $W(x)$] with a smaller timestamp which may still be in transit in the network. When this operation reaches site *i*, it will be rejected since it violates the TO rule: it wants to access a data item that is currently being accessed (in an incompatible mode) by another operation with a higher timestamp.
- It is possible to design an extremely conservative TO algorithm by insisting that the scheduler choose an operation to be sent to the data processor only if there is at least one operation in each queue.
- This guarantees that every operation that the scheduler receives in the future will have timestamps greater than or equal to those currently in the queues.

Multiversion TO Algorithm

- Multiversion TO is another attempt at eliminating the restart overhead cost of transactions.
- Most of the work on multiversion TO has concentrated on centralized databases, so we present only a brief overview.
- We should indicate that multiversion TO algorithm would be a suitable concurrency control mechanism for DBMSs that are designed to support applications that inherently have a notion of versions of database objects (e.g., engineering databases and document databases).
- In multiversion TO, the updates do not modify the database; each write operation creates a new version of that data item. Each version is marked by the timestamp of the transaction that creates it.
- Thus the multiversion TO algorithm trades storage space for time.

- The existence of versions is transparent to users who issue transactions simply by referring to data items, not to any specific version.
- The transaction manager assigns a timestamp to each transaction, which is also used to keep track of the timestamps of each version.
- The operations are processed by the histories as follows:

1. A $R_i(x)$ is translated into a read on one version of x. This is done by finding a version of x (say, x_v) such that $ts(x_v)$ is the largest timestamp less than $ts(T_i)$. $R_i(x_v)$ is then sent to the data processor to read x_v. This case is depicted in

   Figure - a, which shows that R_i can read the version (x_v) that it would have read had it arrived in timestamp order.
2. A $W_i(x)$ is translated into $W_i(x_w)$ so that $ts(x_w) = ts(T_i)$ and sent to the data processor if and only if no other transaction with a timestamp greater than $ts(T_i)$ has read the value of a version of x (say, x_r) such that $ts(x_r) > ts(x_w)$.
   In other words, if the scheduler has already processed a $R_j(x_r)$ such that

   $$ts(T_i) < ts(x_r) < ts(T_j)$$

   then $W_i(x)$ is rejected. This case is depicted in Figure - b, which shows that if W_i is accepted, it would create a version (x_w) that R_j should have read, but did not since the version was not available when R_j was executed – it, instead, read version x_k, which results in the wrong history.

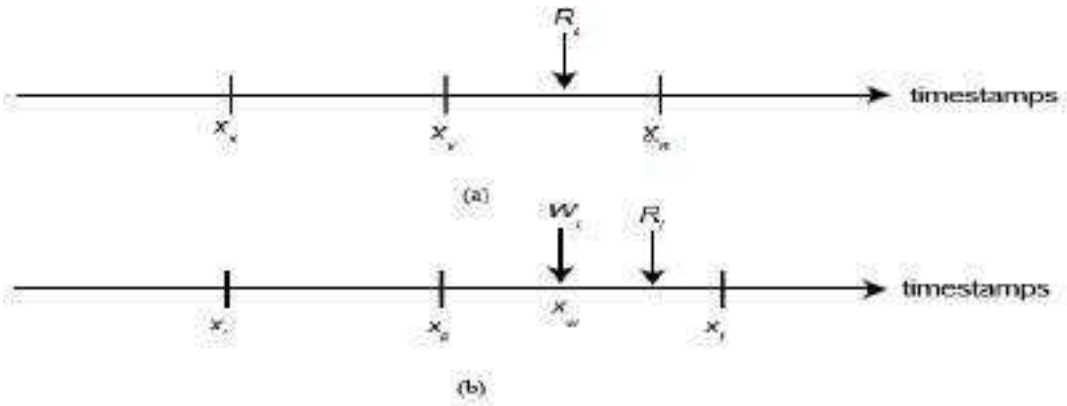


Fig. 11.11 Multiversion TO Cases

- A scheduler that processes the read and the write requests of transactions according to the rules noted above is guaranteed to generate serializable histories.
- To save space, the versions of the database may be purged from time to time. This should be done when the distributed DBMS is certain that it will no longer receive a transaction that needs to access the purged versions.

Optimistic Concurrency Control Algorithms

- The concurrency control algorithms are pessimistic in nature. In other words, they assume that the conflicts between transactions are quite frequent and do not permit a transaction to access a data item if there is a conflicting transaction that accesses that data item.
- The execution of any operation of a transaction follows the sequence of phases: validation (V), read (R), computation (C), write (W) (Figure 11.12). Generally, this sequence is valid for an update transaction as well as for each of its operations.

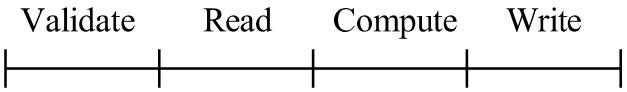


Fig. 11.12 Phases of Pessimistic Transaction Execution

- Optimistic algorithms, on the other hand, delay the validation phase until just before the write phase (Figure 11.13). Thus an operation submitted to an optimistic scheduler is never delayed.
- The read, compute, and write operations of each transaction are processed freely without updating the actual database.
- Each transaction initially makes its updates on local copies of data items. The validation phase consists of checking if these updates would maintain the consistency of the database.
- If the answer is affirmative, the changes are made global (i.e., written into the actual database). Otherwise, the transaction is aborted and has to restart.

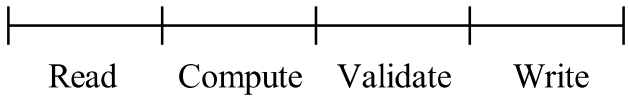


Fig. 11.13 Phases of Optimistic Transaction Execution

- It is possible to design locking-based optimistic concurrency control algorithms.
- The original optimistic proposals are based on timestamp ordering. Therefore, we describe only the optimistic approach using timestamps.
- It differs from pessimistic TO-based algorithms not only by being optimistic but also in its assignment of timestamps. Timestamps are associated only with transactions, not with data items (i.e., there are no read or write timestamps).
- Furthermore, timestamps are not assigned to transactions at their initiation but at the beginning of their validation step. This is because the timestamps are needed only during the validation phase, and as we will see shortly, their early assignment may cause unnecessary transaction rejections.
- Each transaction T_i is subdivided (by the transaction manager at the originating

site) into a number of subtransactions, each of which can execute at many sites. Notationally, let us denote by T_{ij} a subtransaction of T_i that executes at site j.

- Until the validation phase, each local execution follows the sequence depicted in Figure 11.13. At that point a timestamp is assigned to the transaction which is copied to all its subtransactions.
- The local validation of T_{ij} is performed according to the following rules, which are mutually exclusive.

Rule 1. If all transactions T_k where $ts(T_k) < ts(T_{ij})$ have completed their write phase before T_{ij} has started its read phase (Figure 11.14a),[3] validation succeeds, because transaction executions are in serial order.

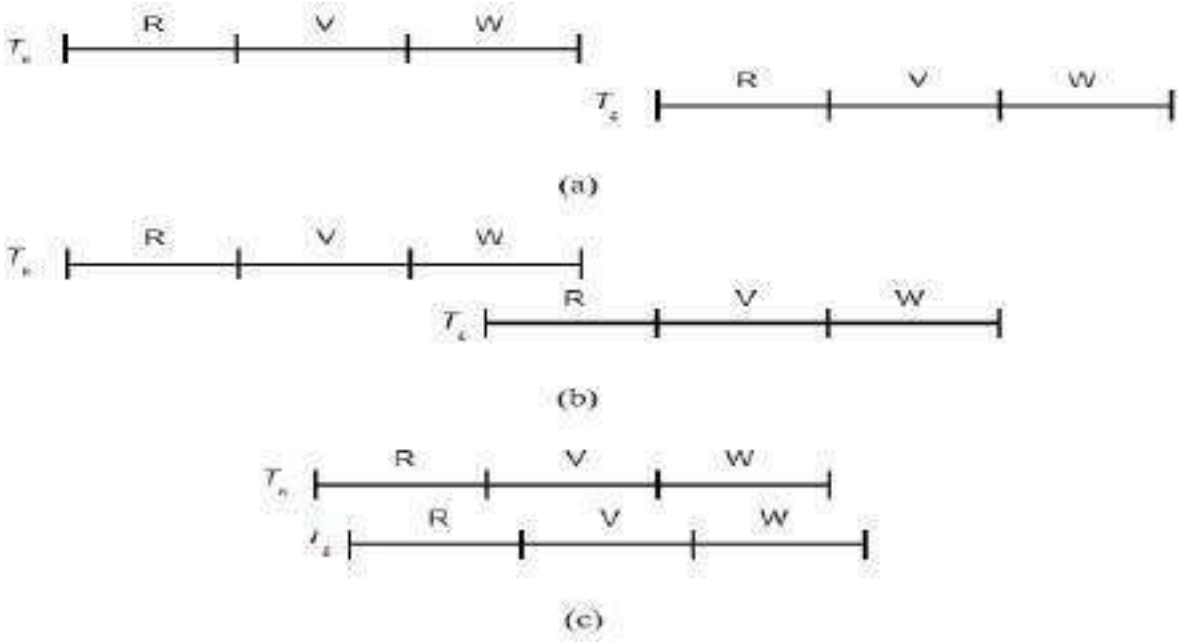


Fig. 11.14 Possible Execution Scenarios

Rule 2. If there is any transaction T_k such that $ts(T_k) < ts(T_{ij})$, and which completes its write phase while T_{ij} is in its read phase (Figure 11.14b), the validation succeeds if $WS(T_k) \cap RS(T_{ij}) = 0/$.

Rule 3. If there is any transaction T_k such that $ts(T_k) < ts(T_{ij})$, and which completes its read phase before T_{ij} completes its read phase (Figure 11.14c), the validation succeeds if $WS(T_k) \cap RS(T_{ij}) = 0/$, and $WS(T_k) \cap WS(T_{ij}) = 0/$.

- Rule 1 is obvious; it indicates that the transactions are actually executed serially in their timestamp order. Rule 2 ensures that none of the data items updated by T_k are read by T_{ij} and that T_k finishes writing its updates into the database before T_{ij} starts writing.
- Thus the updates of T_{ij} will not be overwritten by the updates of T_k. Rule 3 is

similar to Rule 2, but does not require that T_k finish writing before T_{ij} starts writing. It simply requires that the updates of T_k not affect the read phase or the write phase of T_{ij}.

- **An advantage** of optimistic concurrency control algorithms is their potential to allow a higher level of concurrency. It has been shown that when transaction conflicts are very rare, the optimistic mechanism performs better than locking. **A major problem** with optimistic algorithms is the higher storage cost. To validate a transaction, the optimistic mechanism has to store the read and the write sets of several other terminated transactions.
- Another problem is starvation. Consider a situation in which the validation phase of a long transaction fails. In subsequent trials it is still possible that the validation will fail repeatedly.

Deadlock Management

- Any locking-based concurrency control algorithm may result in deadlocks, since there is mutual exclusion of access to shared resources (data) and transactions may wait on locks.
- Some TO-based algorithms that require the waiting of transactions (e.g., strict TO) may also cause deadlocks.
- **A deadlock can occur because transactions wait for one another.**
- A deadlock situation is a set of requests that can never be granted by the concurrency control mechanism.
- ***Example :*** Consider two transactions T_i and T_j that hold write locks on two entities x and y [i.e., $wl_i(x)$ and $wl_j(y)$]. Suppose that T_i now issues a $rl_i(y)$ or a $wl_i(y)$. Since y is currently locked by transaction T_j, T_i will have to wait until T_j releases its write lock on y.
- However, if during this waiting period, T_j now requests a lock (read or write) on x, there will be a deadlock. This is because, T_i will be blocked waiting for T_j to release its lock on y while T_j will be waiting for T_i to release its lock on x. In this case, the two transactions T_i and T_j will wait indefinitely for each other to release their respective locks
- A deadlock is a permanent phenomenon. If one exists in a system, it will not go away unless outside intervention takes place. This outside interference may come from the user, the system operator, or the software system.
- A useful tool in analyzing deadlocks is a *wait-for graph* (WFG). A WFG is a directed graph that represents the wait-for relationship among transactions.
- The nodes of this graph represent the concurrent transactions in the system.

- An edge $T_i \rightarrow T_j$ exists in the WFG if transaction T_i is waiting for T_j to release a lock on some entity.

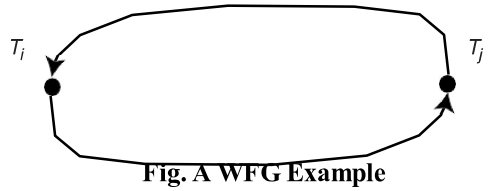
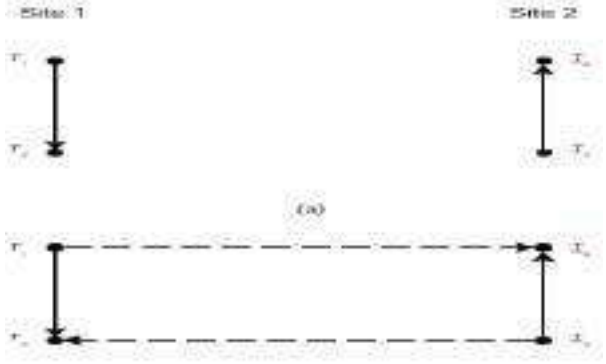


Fig. A WFG Example

- Using the WFG, it is easier to indicate the condition for the occurrence of a deadlock. A deadlock occurs when the WFG contains a cycle.
- We should indicate that the formation of the WFG is more complicated in distributed systems, since two transactions that participate in a deadlock condition may be running at different sites. We call this situation a *global deadlock*.
- In distributed systems, then, it is not sufficient that each local distributed DBMS form a *local wait-for graph* (LWFG) at each site; it is also necessary to form a *global wait-for graph* (GWFG), which is the union of all the LWFGs.

Example : Consider four transactions T_1, T_2, T_3, and T_4 with the following wait- for relationship among them: $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_1$. If T_1 and T_2 run at site 1 while T_3 and T_4 run at site 2, the LWFGs for the two sites are shown in Figure a. Notice that it is not possible to detect a deadlock simply by examining the two LWFGs, because the deadlock is global. The deadlock can easily be detected, however, by examining the GWFG where intersite waiting is shown by dashed lines (Figure b).



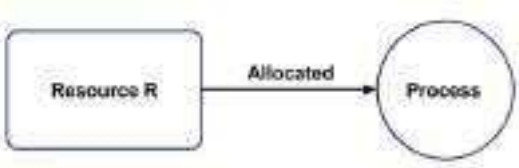
(b)

Fig. Difference between LWFG and GWFG

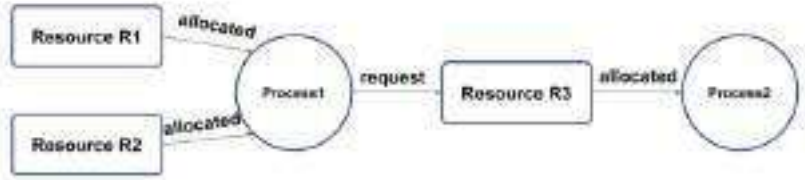
Necessary Conditions of Deadlock

There are four different conditions that result in Deadlock. These four conditions are also known as Coffman conditions and these conditions are not mutually exclusive. Let's look at them one by one.

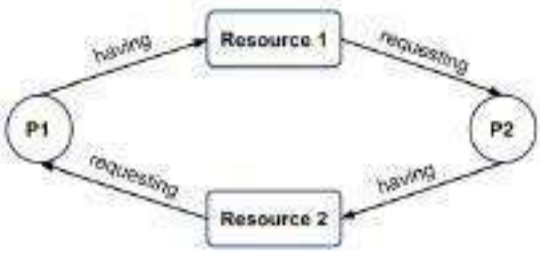
- **Mutual Exclusion:** A resource can be held by only one process at a time. In other words, if a process P1 is using some resource R at a particular instant of time, then some other process P2 can't hold or use the same resource R at that particular instant of time. The process P2 can make a request for that resource R but it can't use that resource simultaneously with process P1.



- **Hold and Wait:** A process can hold a number of resources at a time and at the same time, it can request for other resources that are being held by some other process. For example, a process P1 can hold two resources R1 and R2 and at the same time, it can request some resource R3 that is currently held by process P2.



- **No preemption:** A resource can't be preempted from the process by another process, forcefully. For example, if a process P1 is using some resource R, then some other process P2 can't forcefully take that resource. If it is so, then what's the need for various scheduling algorithm. The process P2 can request for the resource R and can wait for that resource to be freed by the process P1.
- **Circular Wait:** Circular wait is a condition when the first process is waiting for the resource held by the second process, the second process is waiting for the resource held by the third process, and so on. At last, the last process is waiting for the resource held by the first process. So, every process is waiting for each other to release the resource and no one is releasing their own resource. Everyone is waiting here for getting the resource. This is called a circular wait.



- There are three known methods for handling deadlocks:
  - ✓ Deadlock Prevention

- ✓ Deadlock Avoidance
- ✓ Deadlock Detection and Resolution

Deadlock Prevention

- Deadlock prevention methods guarantee that deadlocks cannot occur in the first place.
- Thus the transaction manager checks a transaction when it is first initiated and does not permit it to proceed if it may cause a deadlock.
- To perform this check, it is required that all of the data items that will be accessed by a transaction be predeclared.
- The transaction manager then permits a transaction to proceed if all the data items that it will access are available. Otherwise, the transaction is not permitted to proceed.

- There are two approaches for deadlock prevention:
  1. Non-Preemptive approach (Process not Interrupted)
  2. Preemptive approach (Process can Interrupt)

Deadlock Avoidance

- Deadlock avoidance schemes either employ concurrency control techniques that will never result in deadlocks or require that potential deadlock situations are detected in advance and steps are taken such that they will not occur. We consider both of these cases.
- The simplest means of avoiding deadlocks is to order the resources and insist that each process request access to these resources in that order.
- The lock units in the distributed database are ordered and transactions always request locks in that order. This ordering of lock units may be done either globally or locally at each site.
- Another alternative is to make use of transaction timestamps to prioritize transactions and resolve deadlocks by aborting transactions with higher (or lower) priorities.
- To implement this type of prevention method, the lock manager is modified as follows. If a lock request of a transaction T_i is denied, the lock manager does not automatically force T_i to wait.
- If the test is passed, T_i is permitted to wait for T_j; otherwise, one transaction or the other is aborted.
- Examples of this approach is the WAIT-DIE and WOUND-WAIT algorithms. These algorithms are based on the assignment of timestamps to transactions.

- **WAIT-DIE is a non-preemptive algorithm** in that if the lock request of T_i is denied because the lock is held by T_j, it never preempts T_j, following the rule:

- **WAIT-DIE Rule.** If T_i requests a lock on a data item that is already locked by T_j,

T_i is permitted to wait if and only if T_i is older than T_j. If T_i is younger than T_j, then T_i is aborted and restarted with the same timestamp.

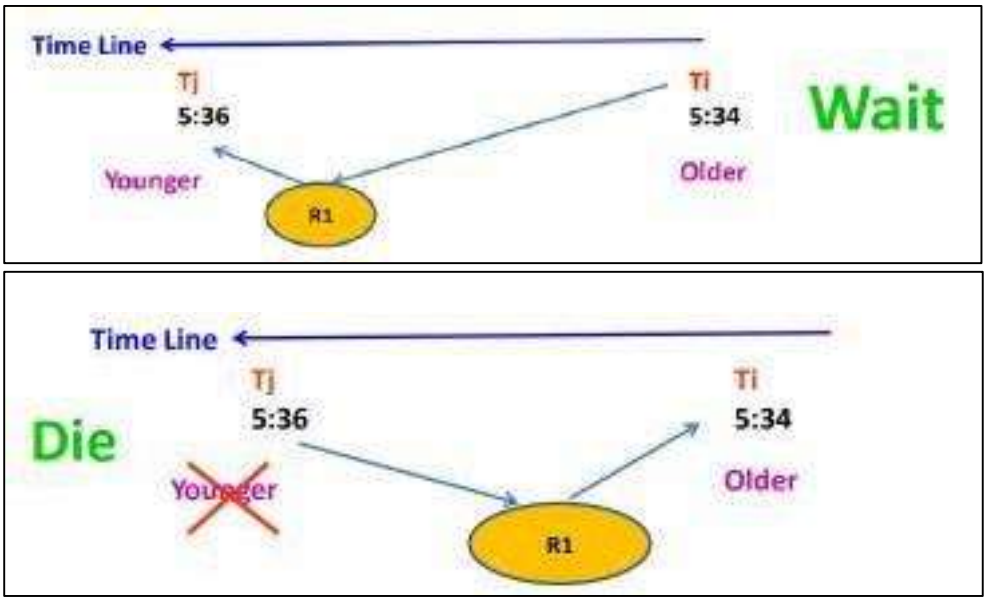


Fig : Non-Preemptive Algorithm (Wait – Die Rule)

- **A preemptive version of the same idea is the WOUND-WAIT algorithm**, which follows the rule:
- **WOUND-WAIT Rule.** If T_i requests a lock on a data item that is already locked by T_j, then T_i is permitted to wait if only if it is younger than T_j; otherwise, T_j is aborted and the lock is granted to T_i.

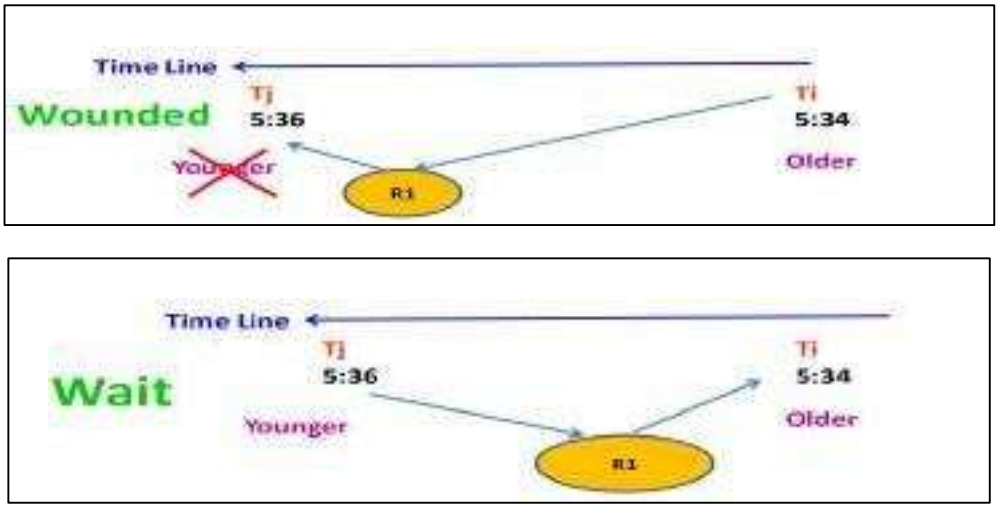


Fig : Preemptive Algorithm (Wound – Wait Rule)

- The rules are specified from the viewpoint of T_i: T_i waits, T_i dies, and T_i wounds T_j. In fact, the result of wounding and dying are the same: the affected transaction is aborted and restarted. With this perspective, the two rules can be specified as follows:

if $ts(T_i) < ts(T_j)$ **then** T_i waits **else** T_i dies (WAIT-DIE)

if $ts(T_i) < ts(T_j)$ **then** T_j is wounded **else** T_i waits (WOUND-WAIT)

- Notice that in both algorithms the younger transaction is aborted. The difference between the two algorithms is whether or not they preempt active transactions.
- Also note that the WAIT-DIE algorithm prefers younger transactions and kills older ones. Thus an older transaction tends to wait longer and longer as it gets older.
- The WOUND-WAIT rule prefers the older transaction since it never waits for a younger one. One of these methods, or a combination, may be selected in implementing a deadlock prevention algorithm.
- Deadlock avoidance methods are more suitable than prevention schemes for timestamp.
- Their fundamental drawback is that they require run-time support for deadlock management, which adds to the run-time overhead of transaction execution.

Deadlock Detection and Resolution

- Deadlock detection and resolution is the most popular and best-studied method. Detection is done by studying the GWFG for the formation of cycles.
- Resolution of deadlocks is typically done by the selection of one or more *victim* transaction(s) that will be preempted and aborted in order to break the cycles in the GWFG.
- Under the assumption that the cost of preempting each member of a set of deadlocked transactions is known, the problem of selecting the minimum total-cost set for breaking the deadlock cycle has been shown to be a difficult (NP-complete) problem. However, there are some factors that affect this choice :

1. The amount of effort that has already been invested in the transaction. This effort will be lost if the transaction is aborted.
2. The cost of aborting the transaction. This cost generally depends on the number of updates that the transaction has already performed.
3. The amount of effort it will take to finish executing the transaction. The scheduler wants to avoid aborting a transaction that is almost finished. To do this, it must be able to predict the future behavior of active transactions (e.g., based on the transaction's type).
4. The number of cycles that contain the transaction. Since aborting a transaction breaks all cycles that contain it, it is best to abort transactions that are part of more than one cycle (if such transactions exist).

- Now we can return to deadlock detection. There are three fundamental 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 the deadlock detector for the entire system. Periodically, each lock manager transmits its LWFG to the deadlock detector, which then forms the GWFG and looks for cycles in it. Actually, the lock managers need only send changes in their graphs (i.e., the newly created or deleted edges) to the deadlock detector.
- The length of intervals for transmitting this information is a system design decision: the smaller the interval, the smaller the delays due to undetected deadlocks, but the larger the communication cost.
- Centralized deadlock detection has been proposed for distributed INGRES. This method is simple and would be a very natural choice if the concurrency control algorithm were centralized 2PL.
- However, the issues of vulnerability to failure, and high communication overhead, must also be considered.

Hierarchical Deadlock Detection

- An alternative to centralized deadlock detection is the building of a hierarchy of deadlock detectors.
- Deadlocks that are local to a single site would be detected at that site using the LWFG. Each site also sends its LWFG to the deadlock detector at the next level. Thus, distributed deadlocks involving two or more sites would be detected by a deadlock detector in the next lowest level that has control over these sites.
- For example, a deadlock at site 1 would be detected by the local deadlock 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 the deadlock involves sites 1 and 4, DD_{0x} detects it, where x is either one of 1, 2, 3, or 4.
- The hierarchical deadlock detection method reduces the dependence on the central site, thus reducing the communication cost.
- It is, however, considerably more complicated to implement and would involve non-trivial modifications to the lock and transaction manager algorithms.

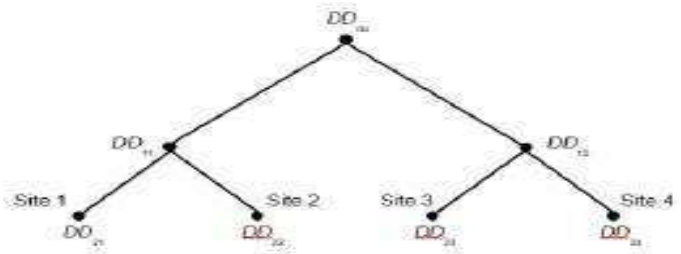


Fig. Hierarchical Deadlock Detection

Distributed Deadlock Detection

- Distributed deadlock detection algorithms delegate the responsibility of detecting deadlocks to individual sites.
- Thus, as in the hierarchical deadlock detection, there are local deadlock detectors at each site that communicate their LWFGs with one another (in fact, only the potential deadlock cycles are transmitted).
- Among the various distributed deadlock detection algorithms, the one implemented in System R* seems to be the more widely known and referenced.
- The LWFG at each site is formed and is modified as follows:

1. Since each site receives the potential deadlock cycles from other sites, these edges are added to the LWFGs.
2. The edges in the LWFG which show that local transactions are waiting for transactions at other sites are joined with edges in the LWFGs which show that remote transactions are waiting for local ones.

- Local deadlock detectors look for two things. If there is a cycle that does not include the external edges, there is a local deadlock that can be handled locally.
- If, on the other hand, there is a cycle involving these external edges, there is a potential distributed deadlock and this cycle information has to be communicated to other deadlock detectors.
  - Obviously, it can be transmitted to all deadlock detectors in the system. In theabsence of any more information, this is the only alternative, but it incurs a high overhead. If, however, one knows whether the transaction is ahead or behind in the deadlock cycle, the information can be transmitted forward or backward along the sites in this cycle.
- The receiving site then modifies its LWFG as discussed above, and checks for deadlocks. Obviously, there is no need to transmit along the deadlock cycle in both the forward and backward directions.
- In the case of, site 1 would send it to site 2 in both forward and backward transmission along the deadlock cycle.
- The distributed deadlock detection algorithms require uniform modification to the lock managers at each site. This uniformity makes them easier to implement. However, there is the potential for excessive message transmission.
- Let the path that has the potential of causing a distributed deadlock in the local WFG of a site be $T_i \rightarrow \cdots \rightarrow T_j$. A local deadlock detector forwards the cycle information only if $ts(T_i) < ts(T_j)$. This reduces the average number of message transmissions by one-half.