BELGIUM CAMPUS iTversity
It's the way we're wired

DATABASE DEVELOPMENT

ARE YOU READY?

DBD371/381

TRANSACTION MANAGEMENT

Transaction Processing in a Distributed System

LEARNING OBJECTIVES

- Transaction Processing in a Distributed System
- Properties of Transactions
- Isolation Levels
- Types of Transactions
- Distributed Concurrency Control
- Taxonomy of Concurrency Control Mechanisms
- Two-Phase Locking (2PL)
- Deadlock Management

TRANSACTION PROCESSING IN A DISTRIBUTED SYSTEM

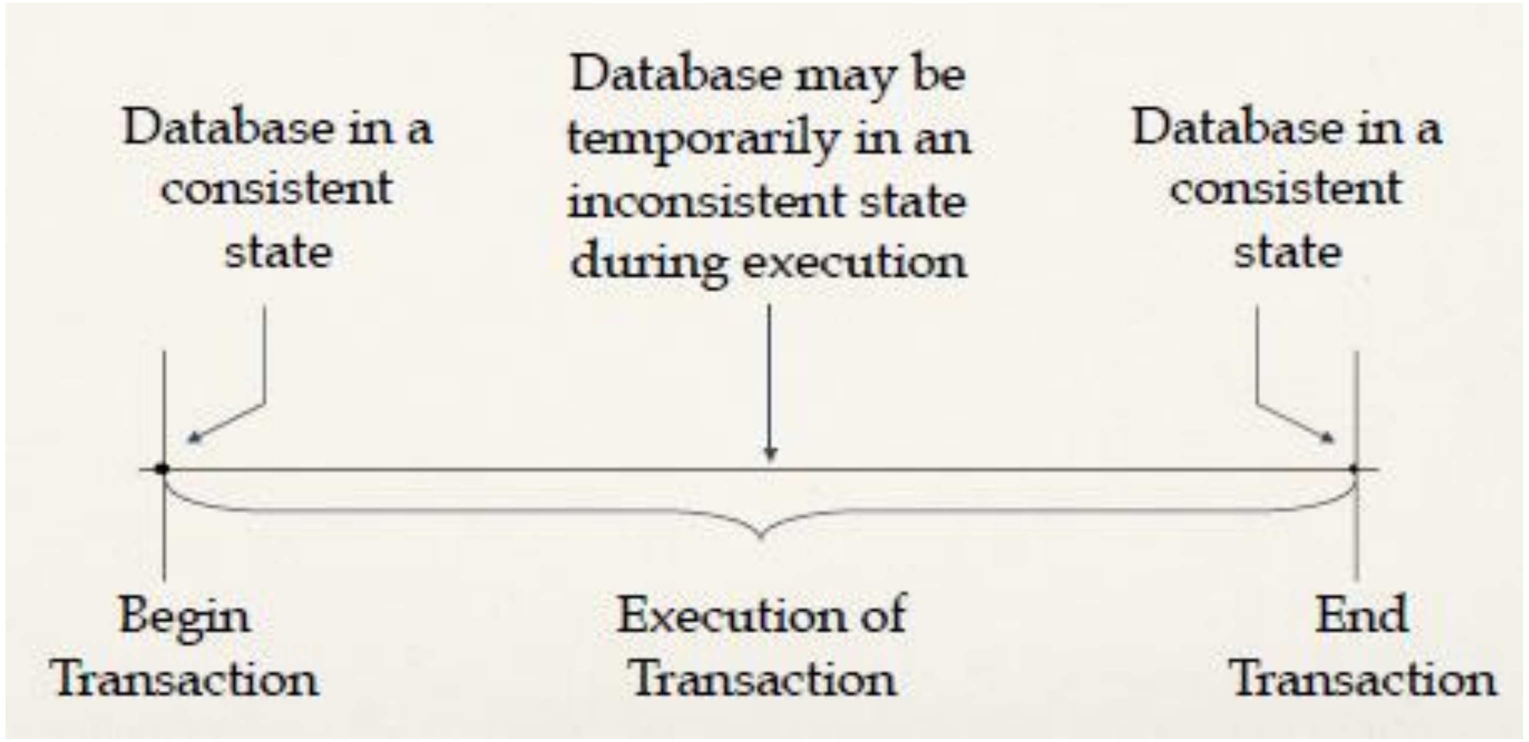
- A transaction is a logical unit of work constituted by one or more SQL statements executed by a single user.
- A transaction begins with the user's first executable SQL statement and ends when it is committed or rolled back by that user
- A **remote transaction** contains only statements that access a single remote node
- A **distributed transaction** contains statements that access more than one node and includes one or more statements that, individually or as a group, update data on two or more distinct nodes of a distributed database.

TRANSACTION MANAGEMENT

- Retrieving-only (or read-only) queries that read data from a distributed database are not a problem
- what happens if, two queries attempt to update the same data item, or if a system failure occurs during execution of a query.
- For retrieve-only queries, neither of these conditions is a problem. One can have two queries reading the value of the same data item concurrently.
- A read-only query can simply be restarted after a system failure is handled.
- For update queries, these conditions can have disastrous effects on the database.
- A transaction is a basic unit of consistent and reliable computing.

TRANSACTION PROCESSING

A transaction is a collection of actions that make consistent transformations of system states while preserving system consistency.



Database in a consistent state
Database may be temporarily in an inconsistent state during execution
Database in a consistent state
Begin Transaction
Execution of Transaction
End Transaction

TRANSACTION MANAGEMENT

- A database is in a consistent state if it obeys all of the consistency (integrity) constraints defined over it
- Transaction consistency, refers to the actions of concurrent transactions.
- The database should remain in a consistent state even if there are a number of user requests that are concurrently accessing (reading or updating)
- A complication arises when replicated databases are considered
- A replicated database is in a mutually consistent state if all the copies of every data item in it have identical values.
- This is referred to as one-copy equivalence since all replica copies are forced to assume the same state at the end of a transaction's execution.
- Transaction management deals with the problems of always keeping the database in a consistent state even when concurrent accesses and failures occur.

Properties of Transactions

- **Atomicity**
  - A transaction is treated as a unit of operation.
  - Either all the transaction's actions are completed, or none of them are.
  - This is also known as the "all-or-nothing property."
- **Consistency**
  - The consistency of a transaction is simply its correctness.
  - A transaction is a correct program that maps one consistent database state to another.
  - Verifying that transactions are consistent is the concern of integrity enforcement,
- **Isolation**
  - Isolation is the property of transactions that requires each transaction to see a consistent database at all times.
  - An executing transaction cannot reveal its results to other concurrent transactions before its commitment.
- **Durability**
  - Ensures that once a transaction commits, its results are permanent and cannot be erased from the database.
  - The dbms ensures that the results of a transaction will survive subsequent system failures.

Consistency
degree of consistency

- Dirty data refers to data values that have been updated by a Transaction prior to its commitment.
- Based on the concept of dirty data, the four levels are defined as:
    1. Degree 3:
    2. Degree 2:
    3. Degree 1
    4. Degree 0:

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 and Degree 1 and 0:

- **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.”

Isolation

- 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.
- (T1 and T2), both of which access data item x. Assume value of x before they start executing is 50.

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

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

Possible Sequence Of Execution

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

No problems here T1 and T2 are executed one after the other and transaction T2 reads 51 as the value of x.
if T1 and T2 are executed one after the other (regardless of the order), the second transaction will read 51

ISOLATION

T2 executes before T1,

Possible Sequence Of Execution

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

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

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

T2 reads 50 as the value of x
. incorrect since T2 reads x while its value is
being changed from 50 to 51

T2's Write will overwrite T1'

Ensuring isolation by not permitting incomplete results to be seen by other transactions, solves the lost updates problem. (cursor stability Isolation)

cascading aborts.→ reason for isolation. Transaction that has read incomplete values will have to abort as well

Degrees of Isolation among transactions

- **Degree 0** provides very little isolation other than preventing lost updates.
- **Degree 1** T does not overwrite dirty data of other transactions and does not commit any writes before EOT
- **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.(Strict)

Isolation Levels

- **According to Isolation property, each transaction should see a consistent database at all times.**
- **No other transaction can read or modify data that is being modified by another transaction.**
- **If this property is not maintained:**
  - ➢ **Dirty read:**
  - ➢ **Non-repeatable (fuzzy) read:**
  - ➢ **Phantom**
  - ➢ **snapshot isolation**

DIRTY READ:

- **Refers to data items whose values have been modified by a transaction that has not yet committed.**
- T1 modifies x which is then read by T2 before T1 terminates; T1 aborts ⇒ T2 has read value which never exists in the database.

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

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

Non-repeatable (fuzzy) read:

- T1 reads x; T2 then modifies or deletes x and commits. T1 tries to read x again but reads a different value or can't find it.
- Two reads within the same transaction T1 return different

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

or

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

Phantom

- ➞ T1 searches the database according to a predicate while T2 inserts new tuples that satisfy the 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

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

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 –
- the writes are not visible to other transactions and it does not see the writes of other transactions.

Consistency levels.

- **Read Uncommitted:** For transactions operating at this level, all three phenomena are possible. (Dirty read, Non-repeatable (fuzzy) read, **Phantom)**
- **Read Committed : Fuzzy reads** and **phantoms** are possible, but dirty reads are not.
- **Repeatable Read:** Only **phantoms** possible.
- **Anomaly serializable**: None of the phenomena are possible.

Types of Transactions

- Transactions have been classified according to a number of criteria.
- **1. Duration of transactions**
  - **Online (short-life)** covers a large majority of current transaction applications
  - **batch** (Long-life) take longer to execute (response time and access a larger portion of the database.
  - **conversational**, which is executed by interacting with the user issuing it.

Organization of the read and write actions

BELGIUM CAMPUS iTversity
It's the way we're wired

1. **general:** intermix their read and write actions without any **specific ordering.**

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

1. **two-step transaction:** the transactions are restricted so that all the read actions are performed before any write action.

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

2. **restricted (or read-before-write):**the transaction is restricted so that a data item has to be read before it can be updated (written),

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

3. **restricted two-step transaction** a transaction is both two-step and restricted,

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

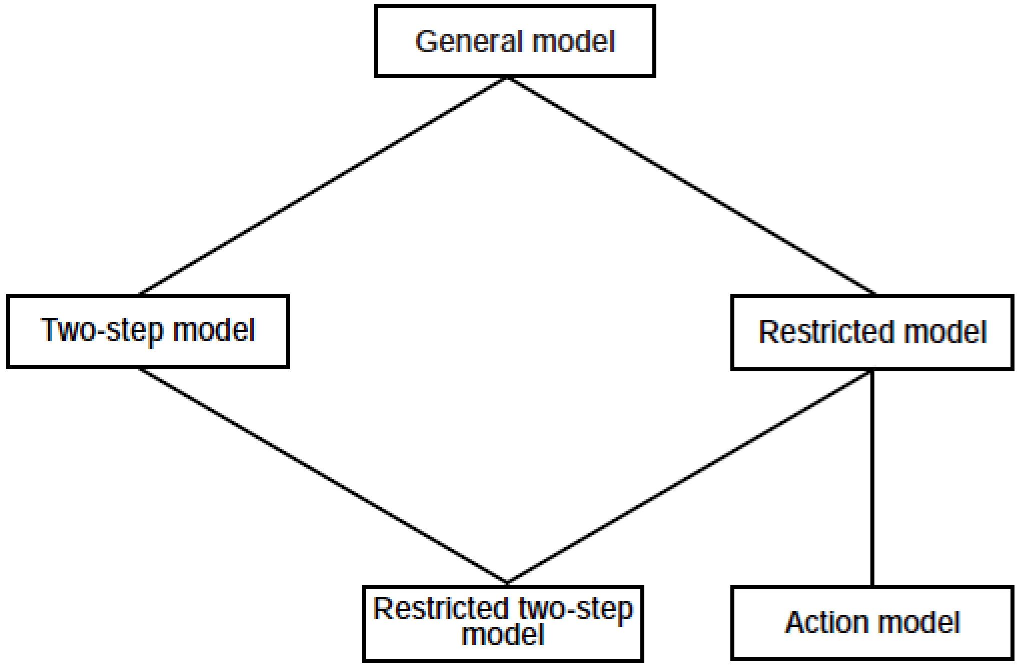
4. **Action model:** consists of the restricted class with the further restriction that each (read, write) pair be executed atomically.

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

Each pair of actions within square brackets is executed atomically

CONCURRENCY CONTROL BY MULTIPLE VERSIONS.

Transaction Models



Generality increases upward.

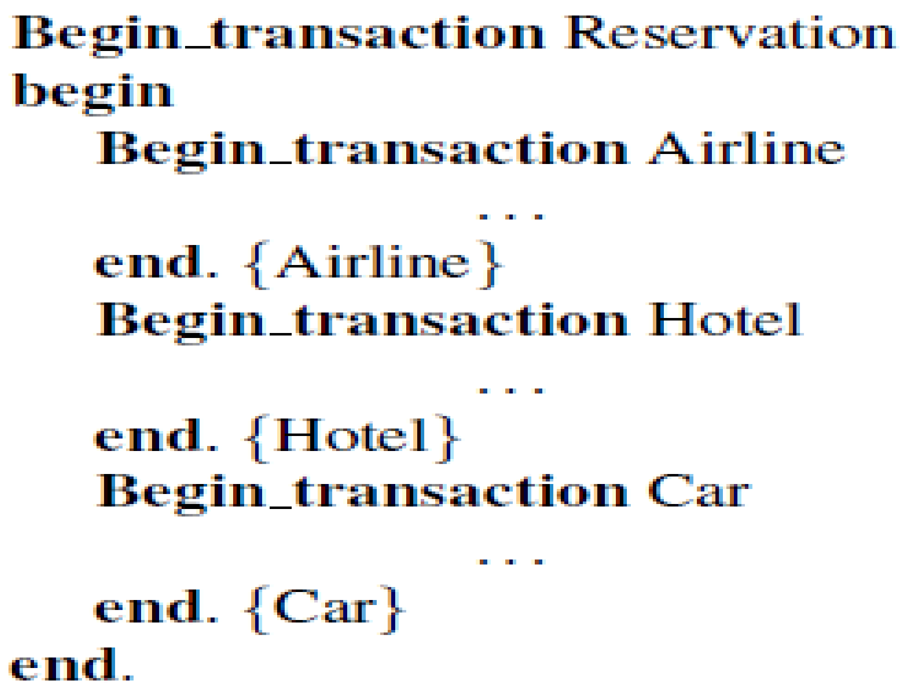
CATEGORIES IN THE STRUCTURES OF TRANSACTIONS

- Transactions can also be classified according to their structure
- Categories:

1. **Flat transactions**, (single start point (Begin transaction) and a single termination point (End transaction).)
2. **nested transactions**: include other transactions with their own begin and commit points. (
   - **Closed** → begins after its parent and finishes before it, and the commitment of the subtransactions is conditional upon the commitment of the parent.)
   - Open → (allows its partial results to be observed outside the transaction.
3. **Workflow models** : activity consisting of a set of tasks with well-defined precedence relationship among them.

Nested Transactions

- Include other transactions with their own begin and commit points

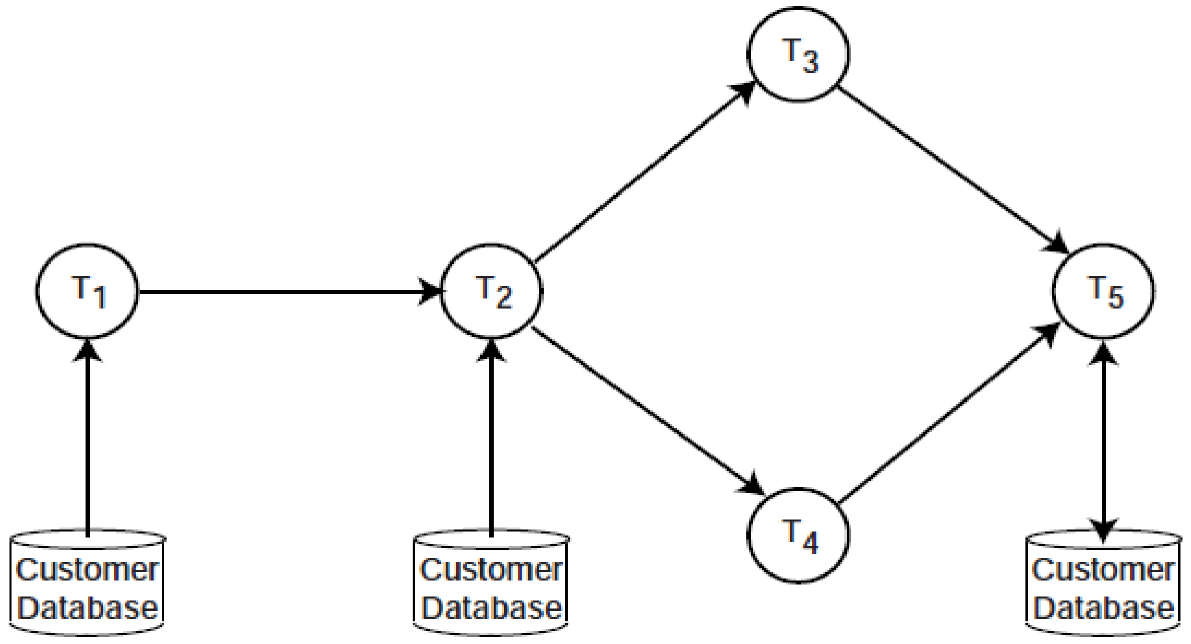


```
Begin_transaction Reservation
begin
   Begin_transaction Airline
               . . .
   end. {Airline}
   Begin_transaction Hotel
               . . .
   end. {Hotel}
   Begin_transaction Car
               . . .
   end. {Car}
end.
```

Workflows

- Workflow is “a collection of tasks organized to accomplish some business process.”
- Three types of workflows
  - **Human-oriented workflows**, which involve humans in performing the tasks.
  - **System-oriented workflows** are those that consist of computation-intensive and specialized tasks that can be executed by a computer.
  - **Transactional workflows** range in between human-oriented and system oriented workflows and borrow characteristics from both. involve “coordinated execution of multiple tasks that
    - (a) may involve humans,
    - (b) require access to HAD [heterogeneous, autonomous, and/or distributed] systems,
    - (c) support selective use of transactional properties [i.e., ACID properties] for individual tasks or entire workflows.” [
    - The common point among them is that activity consists of a set of tasks with well-defined precedence relationship among them.

RESERVATION TRANSACTION EXAMPLE



- Customer request is obtained (task T1) and Customer Database is accessed to obtain customer information, preferences, etc.;
- Airline reservation is performed (T2) by accessing the Flight Database;
- Hotel reservation is performed (T3), which may involve sending a message to the hotel involved;
- Auto reservation is performed (T4), which may also involve communication with the car rental company;
- Bill is generated (T5) and the billing info is recorded in the billing database.

In this workflow there is a serial dependency of T2 on T1,and T3, T4 on T2; however, T3 and T4 (hotel and car reservations) are performed in parallel and T5 waits until their completion.

A workflow is modelled as an activity with open nesting semantics in that it permits partial results to be visible outside the activity boundaries.

THE TRANSACTION CONCEPT

- The distributed execution monitor consists of two modules:
  1. **Transaction manager (TM) →** responsible for coordinating the execution of the database operations on behalf of an application.
  2. **Scheduler (SC). →**responsible for the implementation of a specific concurrency control algorithm for synchronizing access to the database.

- The local recovery managers (LRM) that exist at each site. implement the local procedures by which the local database can be recovered to a consistent state following a failure.
- Each transaction originates at one site, **(originating site.)**
  - ➢The execution of the database operations of a transaction is coordinated by the TM at that transaction's originating site

DISTRIBUTED CONCURRENCY CONTROL

- 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, as defined is maintained in a multiuser distributed environment.
- If transactions are internally consistent, the simplest way of achieving this objective is to execute each transaction alone, one after another.
- The level of concurrency is 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.

Concurrency control mechanisms

1. Serializability
2. locking-based(pessimistic algorithms; optimistic concurrency control)
3. timestamp ordering-based

Isolation

- Consider the following two transactions:

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

- Possible execution sequences:

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

SERIALIZABILITY THEORY

- If the concurrent execution of transactions leaves the database in a state that can be achieved by their serial execution in some order, problems such as lost updates will be resolved.(**serializability**).
- A history R (also called a schedule) is defined over a set of transactions

  T = T1;T2; : : : ;Tn and specifies an interleaved order of execution of these transactions' operations.
- Based on the definition of a transaction the history can be specified as a partial order over T.
- A history, defines the execution order of all operations in its domain.

CONDITIONS APPLICABLE

Three conditions are applicable:

1. Condition 1 states that the domain of the history is the union of the domains of individual transactions.
2. Condition 2 states that the ordering relation of the history is a superset of the ordering relations of individual transactions. This maintains the ordering of operations within each transaction.
3. Condition 3 simply defines the execution order among conflicting operations in H.

CONDITIONS APPLICABLE

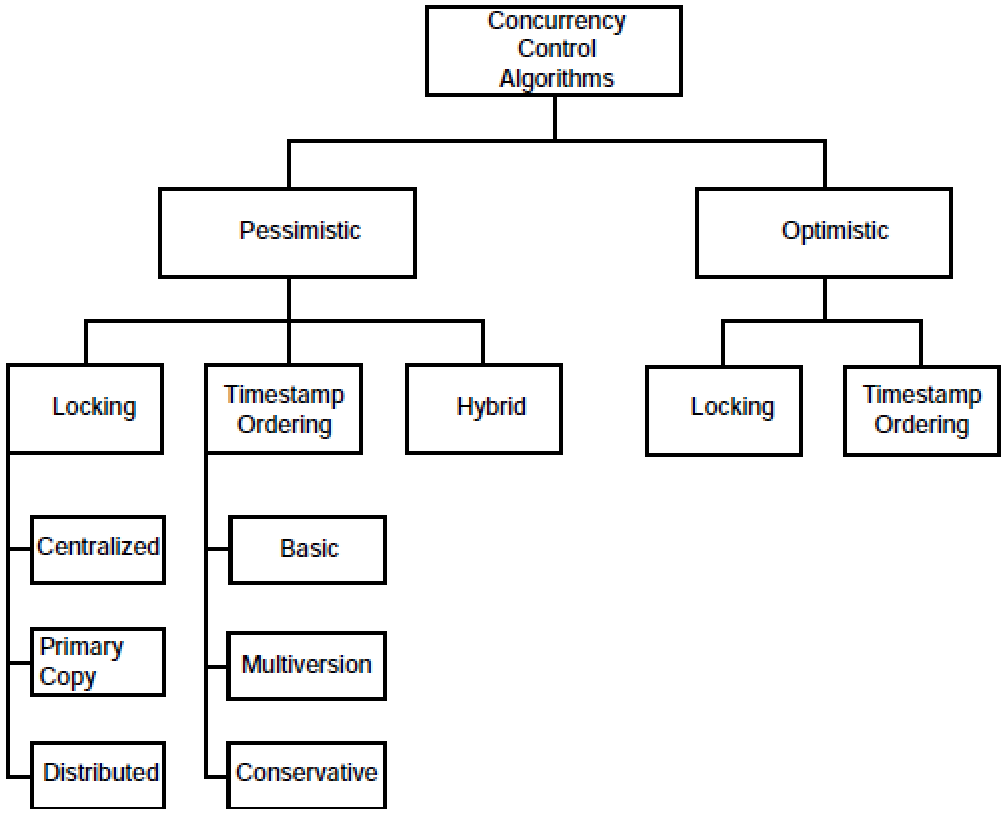
- A history H is said to be serializable if and only if it is conflict equivalent to a serial history
- Two histories H1 and H2, defined over the same set of transactions T, are equivalent if they have the same effect on the database.
- The primary function of a concurrency controller is to generate a serializable history for the execution of pending transactions.
- The issue, is to devise algorithms that are guaranteed to generate only serializable histories.
- In a distributed database 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.

Taxonomy of Concurrency Control Mechanisms

- A number of ways that the concurrency control approaches can be classified
- Group the concurrency control mechanisms into two broad classes:
    1. pessimistic concurrency control methods
    2. optimistic concurrency control methods.

1. **Pessimistic algorithms** synchronize the concurrent execution of transactions early in their execution life cycle, consists of locking based algorithms, ordering (or transaction ordering) based algorithms, and hybrid algorithms.
2. **optimistic algorithms** delay the synchronization of transactions until their termination and classified as locking-based or timestamp ordering-based

TAXONOMY OF CONCURRENCY CONTROL MECHANISMS

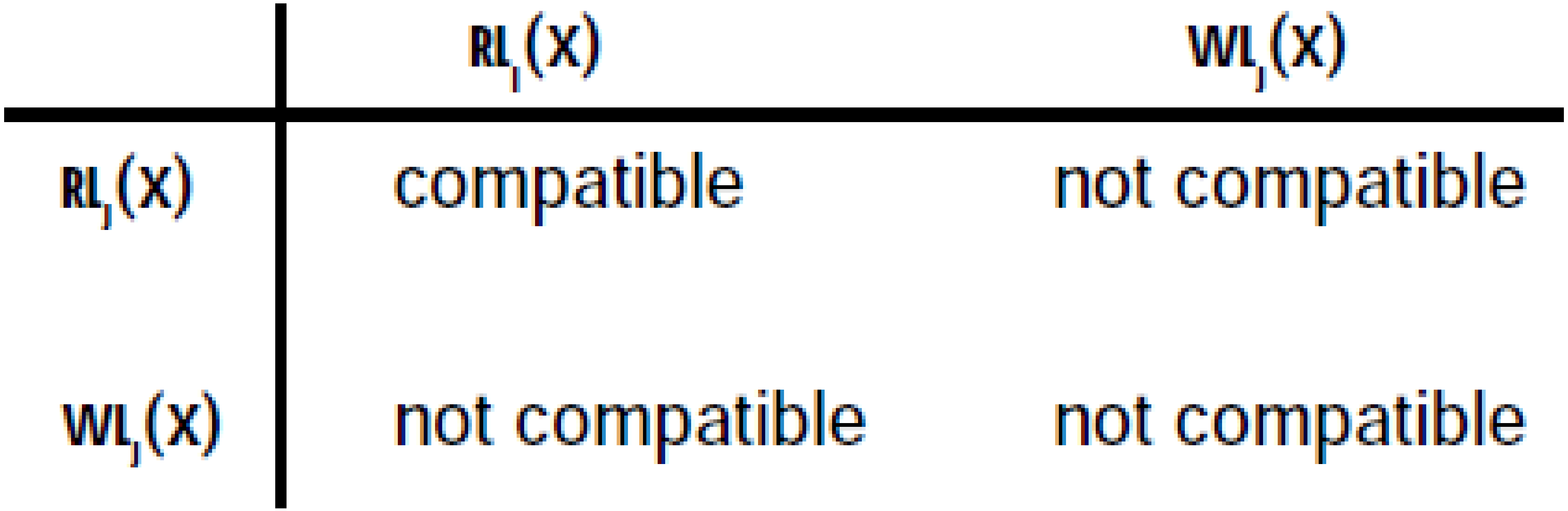


Locking-Based Concurrency Control Algorithms

- 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.
- Two types of locks (lock modes) associated with each lock unit:
    1. read lock (rl)
    2. write lock (wl).
- 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.
- Distributed DBMS handles the lock management responsibilities on behalf of the transactions

LOCKING-BASED CONCURRENCY CONTROL ALGORITHMS

COMPATIBILITY MATRIX OF LOCK MODES

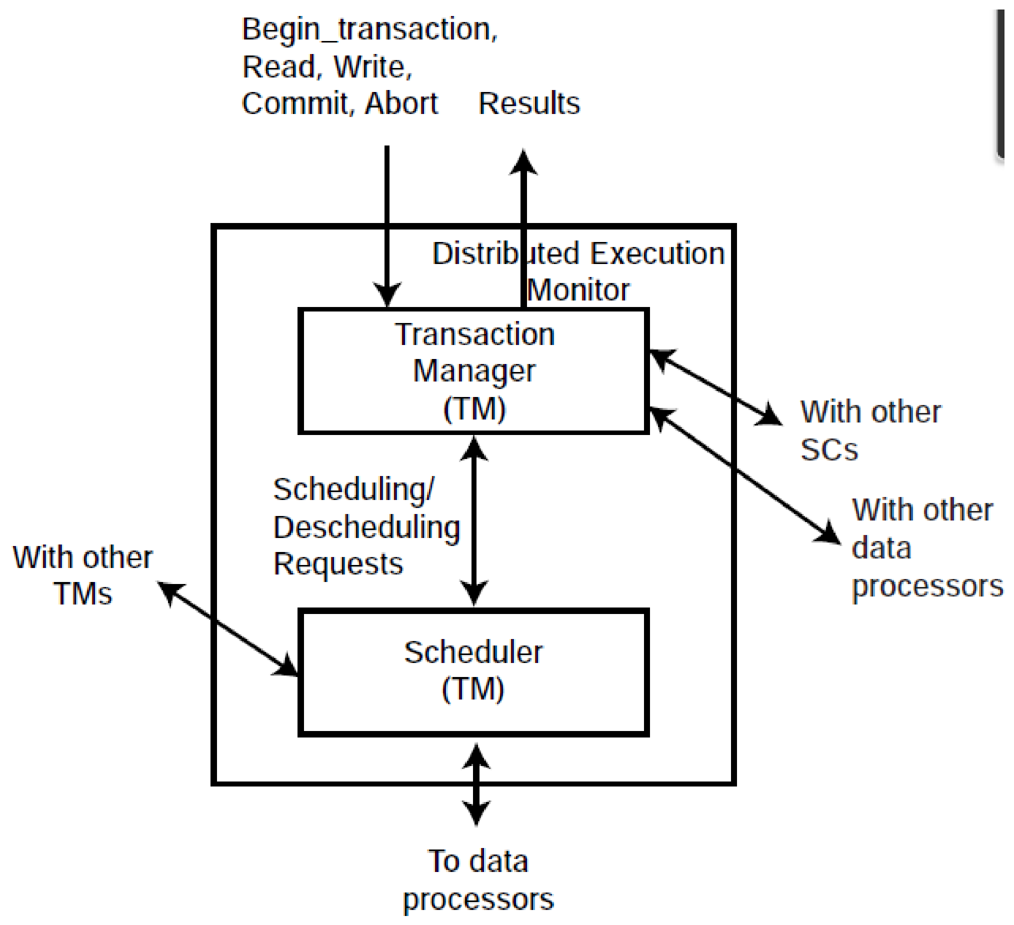


	$RL_i(x)$	$WL_i(x)$
$RL_j(x)$	compatible	not compatible
$WL_j(x)$	not compatible	not compatible

LOCKING-BASED CONCURRENCY CONTROL ALGORITHMS

- In locking-based systems, the scheduler is a lock manager (LM).
- The **Transaction manager** passes to the **Lock manager** the database operation and associated information
- The **Lock manager** then checks if the lock unit that contains the data item is already locked.
- If so, and **if the existing lock mode is incompatible** with that of the current transaction, the current operation is delayed.
- Otherwise, the lock is set in the desired mode and the database **operation is passed** on to the **Data Processor** for actual database access.
- The **Transaction manager** is then informed of the results of the operation.
- The **termination of a transaction** results in the **release of its locks** and the **initiation of another transaction** that might be waiting for access to the same data item.
- **To synchronize transaction** executions and **generate serializable histories**, the locking and releasing operations of transactions also need to be coordinated

LOCKING-BASED CONCURRENCY CONTROL ALGORITHMS



Begin_transaction, Read, Write, Commit, Abort
Results
Distributed Execution Monitor
Transaction Manager (TM)
With other SCs
With other data processors
Scheduling/ Descheduling Requests
With other TMs
Scheduler (TM)
To data processors

Concurrency control is the problem of synchronizing concurrent transactions (i.e., order the operations of concurrent transactions) such that the following two properties
are achieved:

1. – the consistency of the DB is maintained
2. – the maximum degree of concurrency of operations is achieved

- Obviously, the serial execution of a set of transactions achieves consistency, if every single transaction is consistent

EXAMPLE

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

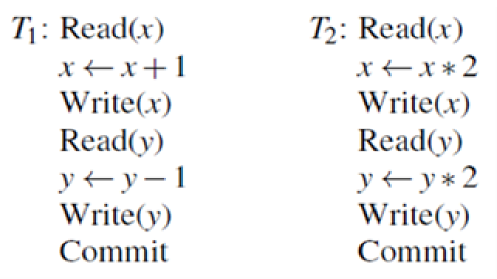
T_2: Read(x)
$x \leftarrow x * 2$
Write(x)
Read(y)
$y \leftarrow y * 2$
Write(y)
Commit

What is The problem with history H in this Example ?

Valid history that a lock manager employing the locking Algorithm may generate

H ={wl1(x);R1(x);W1(x)**;lr1(x);wl2(x);**R2(x);w2(x)**;lr2(x);wl2(y);**R2(y);W2(y)**;lr2(y);wl1(y);**R1(y);W1(y); **lr1(y)}**

IS H a serializable history ? Check using the values x and y as 50 and 20, respectively



T_1:	T_2:
Read(x)	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

Valid history that a lock manager employing the locking Algorithm may generate

H ={wl1(x);R1(x);W1(x);lr1(x);wl2(x);R2(x);w2(x);lr2(x);wl2(y);R2(y);W2(y);lr2(y);wl1(y);R1(y);W1(y); lr1(y)}

READ(X) = 50	READ(X) = 51	R1(X) = 50
X= X+ 1 = 51	X= X*2 =102	W1(X) = 51
		R2(X) =51
WRITE(X) = 51	WRITE(X) 102	W2(X) =102
READ(Y) = 20	READ(Y) = 19	R2(Y) = 20
Y= Y-1 = 19	Y = Y*2 = 38	W2(Y) = 40
WRITE(Y) 19	WRITE(Y) = 38	R1(Y) = 40
		W1(Y) = 39 H NOT SERIALIZABLE

TEST FOR SERIALIZABILITY AND CONFLICT EQUIVALENCE

- Conflict Equivalent: Two schedules are said to be conflict equivalent when one can be transformed to another by swapping non-conflicting operations.
- T1: R1(A), W1(A), R1(B), W1(B)
- T2: R2(A), W2(A), R2(B), W2(B)
- **S1: R1(A), W1(A), R2(A), W2(A), R1(B), W1(B), R2(B), W2(B)**
- **S11: R1(A), W1(A), R1(B), W2(A), R2(A), W1(B), R2(B), W2(B)**

TEST FOR SERIALIZABILITY AND CONFLICT EQUIVALENCE

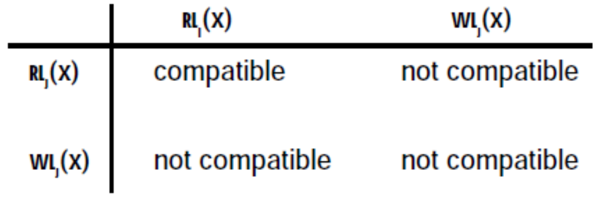
- **S1: R1(A), W1(A), R2(A), W2(A), R1(B), W1(B), R2(B), W2(B)**
- **S11: R1(A), W1(A), R1(B), W2(A), R2(A), W1(B), R2(B), W2(B)**
- **S1: R1(A), W1(A), R1(B), W2(A), R2(A), W1(B), R2(B), W2(B)**
- Swap R2(A) and R1(B) in S1
- **R1(A), W1(A), R1(B), W2(A), R2(A), W1(B), R2(B), W2(B)**
- Swap $W_2(A)$ and $W_1(B)$
- **R1(A), W1(A), R1(B), W1(B), R1(B), R2(A), W2(A), W2(B)**
- Since S has been transformed into a serial schedule by swapping non-conflicting operations $W_2(A)$ and $W_1(B)$ **S1** is serializable.

TEST FOR SERIALIZABILITY AND CONFLICT EQUIVALENCE

- Two transactions :
- T1: R1(A), W1(A), R1(B), W1(B)
- T2: R2(A), W2(A), R2(B), W2(B)
- S2: R2(A), W2(A), R1(A), W1(A), R1(B), W1(B), R2(B), W2(B)
- Is S2 Serializable?
- swap R1(A) and R2(B)
- R2(A), W2(A), R2(B), W1(A), R1(B), W1(B), R1(A), W2(B)
- Swap W1(A), and W2(B)
- R2(A), W2(A), R2(B), W2(B), R1(B), W1(B), R1(A), W1(A)

TEST FOR SERIALIZABILITY

- S2: R2(A), W2(A), R1(A), W1(A), R1(B), W1(B), R2(B), W2(B)
- Is S2 Serializable?
- Swap R1(A) and R2(B)
  - R2(A), W2(A), R2(B), W1(A), R1(B), W1(B), R1(A), W2(B)
- Swap W1(A), and W2(B)
  - R2(A), W2(A), R2(B), W2(B), R1(B), W1(B), R1(A), W1(A)



	$RL_i(X)$	$WL_j(X)$
$RL_j(X)$	compatible	not compatible
$WL_j(X)$	not compatible	not compatible

TWO-PHASE LOCKING (2PL)

- The two-phase locking(2PL) rule states that no transaction should request a lock after it releases one of its locks. i.e a transaction should not release a lock until it is certain that it will not request another lock.

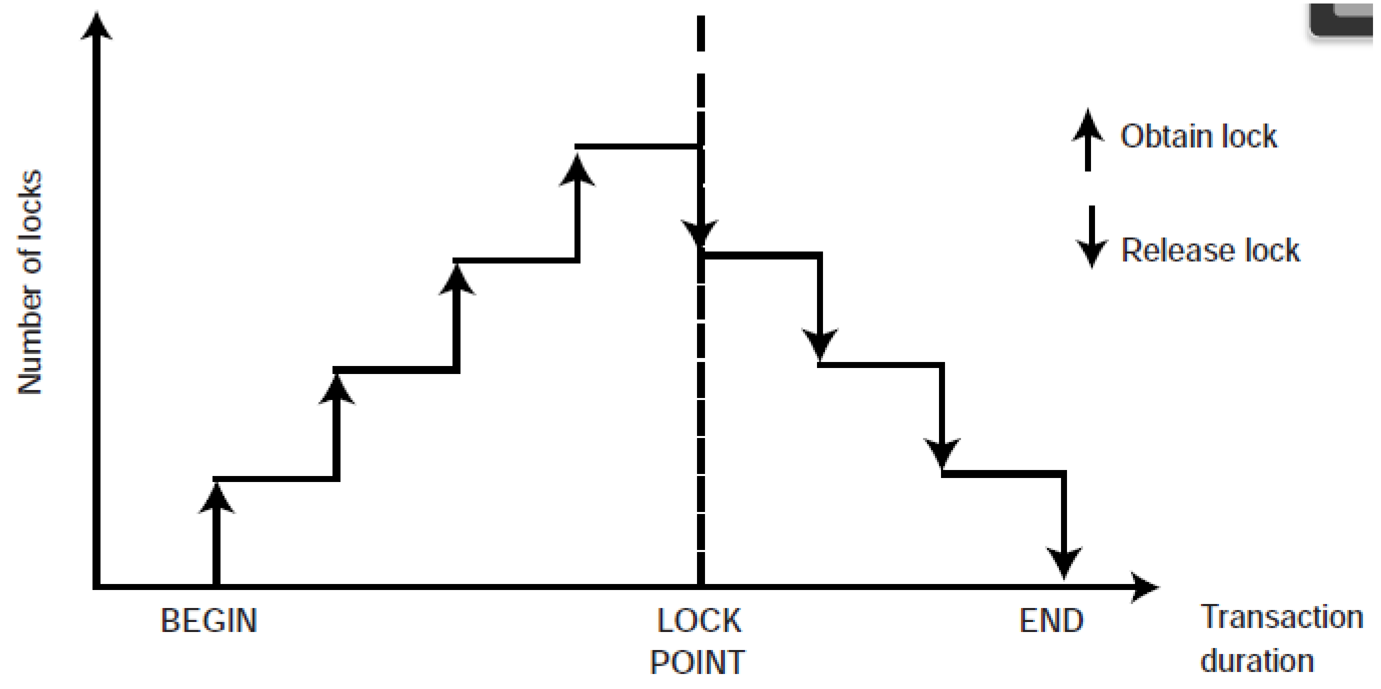
Two-Phase Locking (2PL)

- 2PL algorithms execute transactions in two phases.

1. Each transaction has a growing phase, where it obtains locks and accesses data items,
2. A shrinking phase, during which it releases locks

2PL Lock Graph

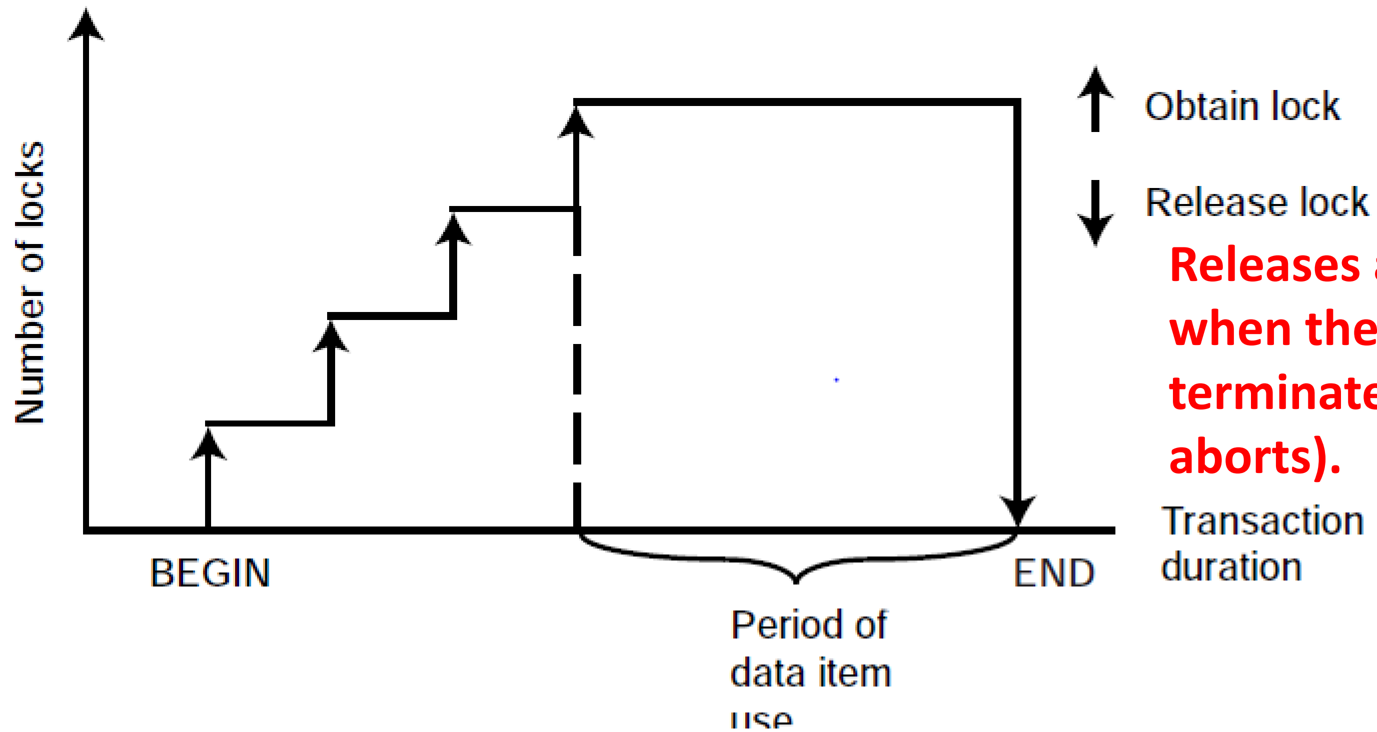
lock manager releases locks as soon as access to that data item has been completed.
What is the problem with this?



TWO-PHASE LOCKING (2PL)

- 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

STRICT 2PL LOCK GRAPH

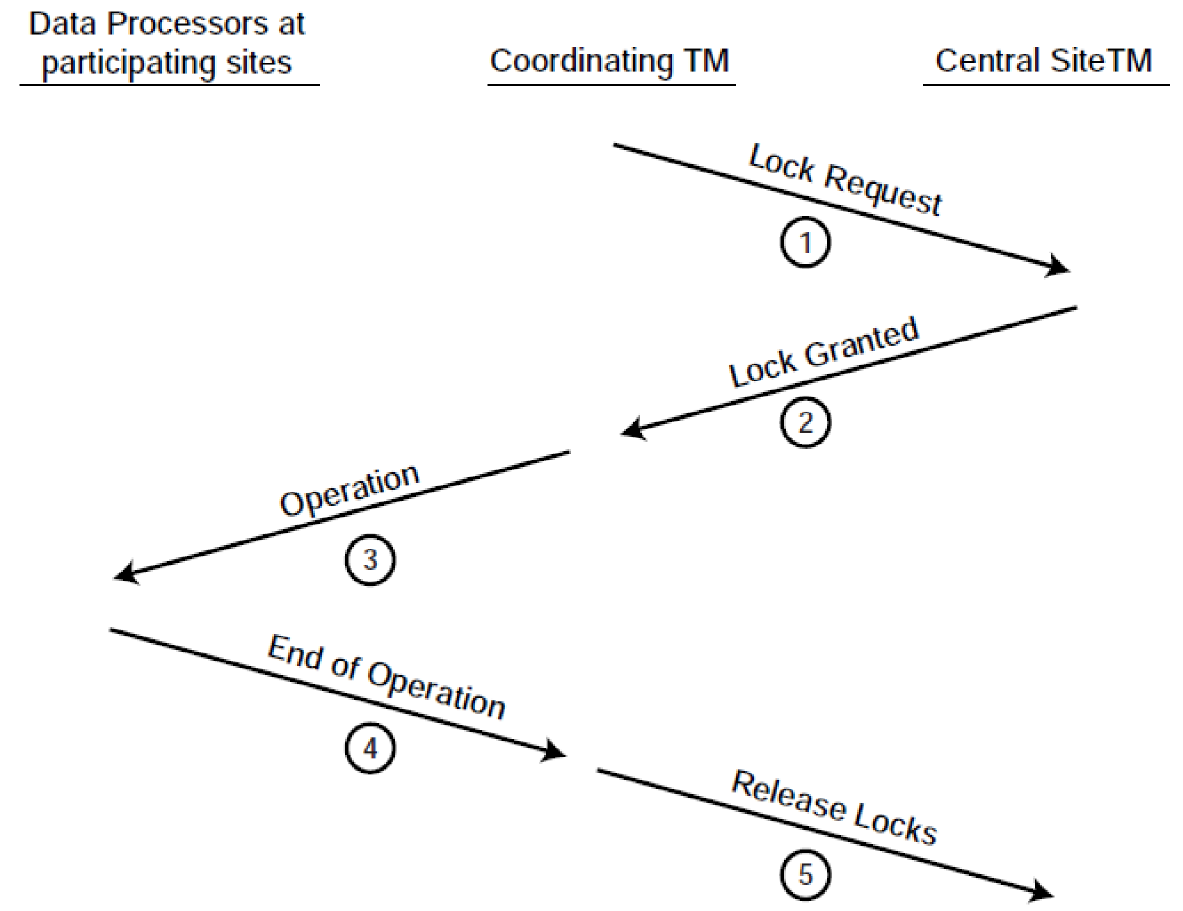


Releases all the locks together when the transaction terminates (commits or aborts).

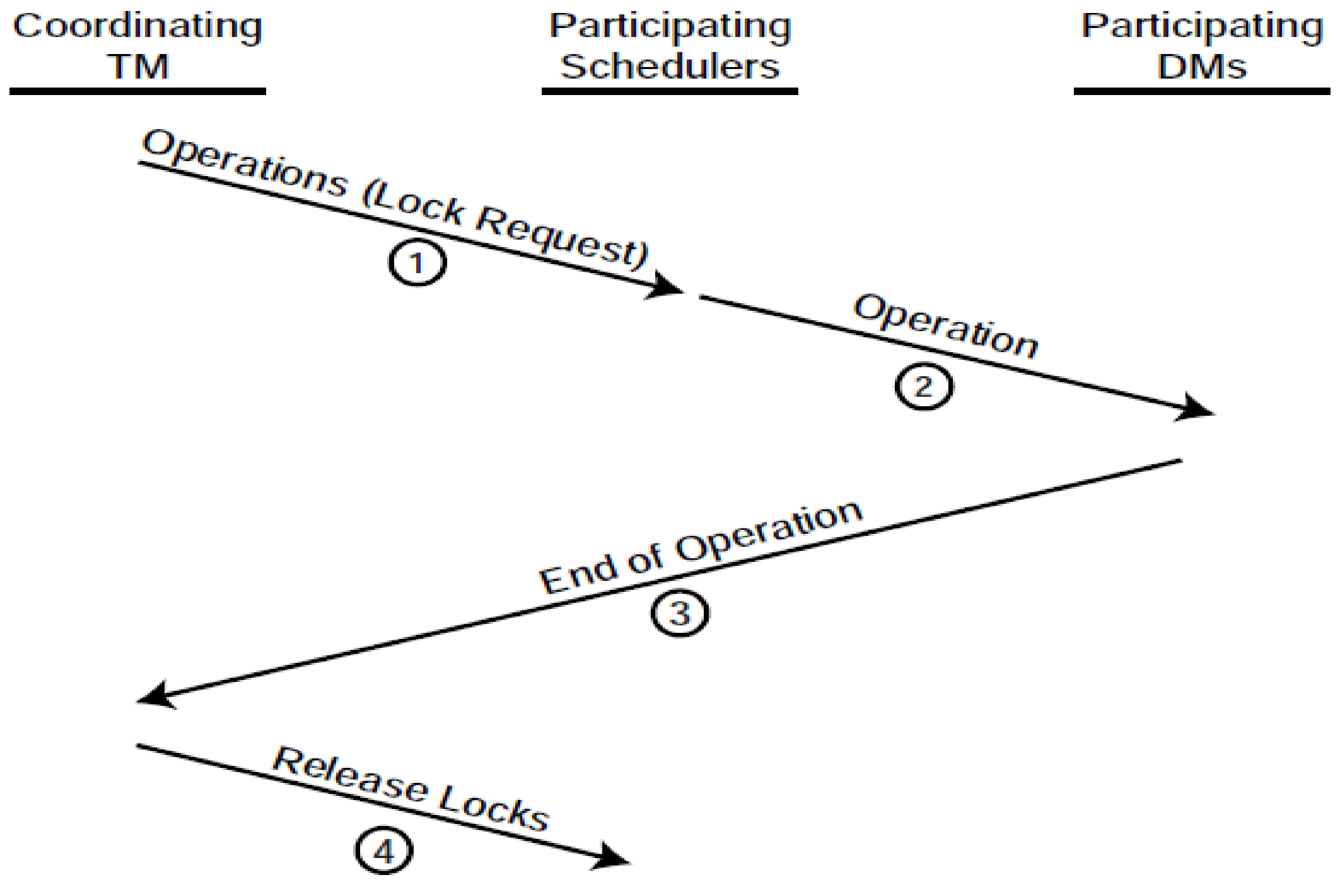
Centralized 2PL

- The 2PL algorithm can easily be extended to the distributed DBMS environment.
- One way of doing this is to delegate lock management responsibility to a single site only.
- 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

Communication Structure of Centralized 2PL



Distributed 2PL



Distributed 2PL (D2PL) requires the availability of lock managers at each site

BELGIUM CAMPUS iTversity
It's the way we're wired

Distributed 2PL

- The distributed 2PL transaction management algorithm is similar to the C2PL-TM, with two major modifications.
  1. 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.
  2. 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.
- The participating data processors send the “end of operation” messages to the coordinating TM.

TIMESTAMP-BASED CONCURRENCY CONTROL ALGORITHMS

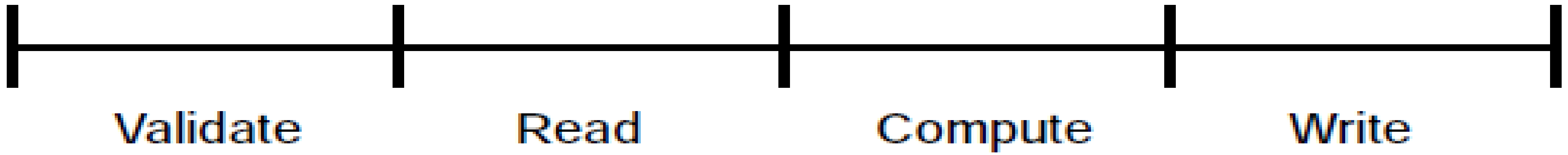
- Timestamp-based concurrency control algorithms do not attempt to maintain serializability by mutual exclusion.
- Instead, they select, a priori, a serialization order and execute transactions accordingly.
- The transaction manager assigns each transaction T_i a unique timestamp,ts(Ti), at its initiation.
- A timestamp is a simple identifier that serves to identify each transaction uniquely and is used for ordering.
- There are a number of ways that timestamps can be assigned. One method is to use a global (system-wide) increasing counter.
- The maintenance of global counters is a problem in distributed systems. Therefore, it is preferable that each site autonomously assigns timestamps based on its local counter.
- To maintain uniqueness, each site appends its own identifier to the counter value.

Basic TO Algorithm

- In the basic TO algorithm :
  - ➢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. (ensures that the transaction has a chance to execute in its next try.)
  - ➢basic TO algorithm never causes deadlocks. the penalty of deadlock freedom is potential restart of a transaction numerous times.

Pessimistic Concurrency Control Algorithms

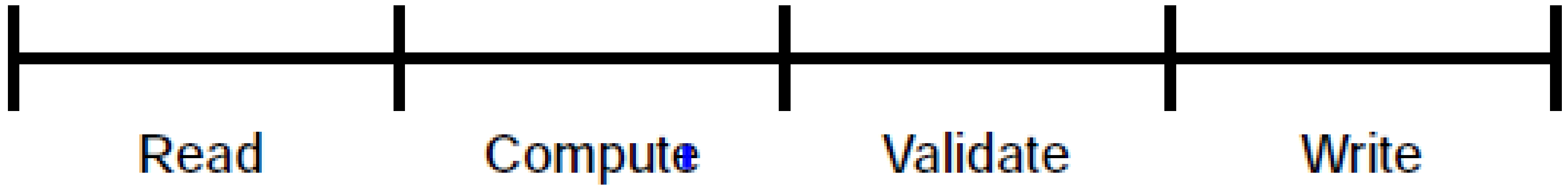
- Pessimistic concurrency control algorithms assume that the conflicts between transactions are quite frequent and do not permit a transaction to access a data item if there is a conflicting transaction that accesses that data item.
- The execution of any operation of a transaction follows the sequence of phases: validation (V), read (R), computation (C), write (W)



Phases of Pessimistic Transaction Execution

Optimistic Concurrency Control Algorithms

- Optimistic algorithms, on the other hand, delay the validation phase until just before the write phase
- 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.



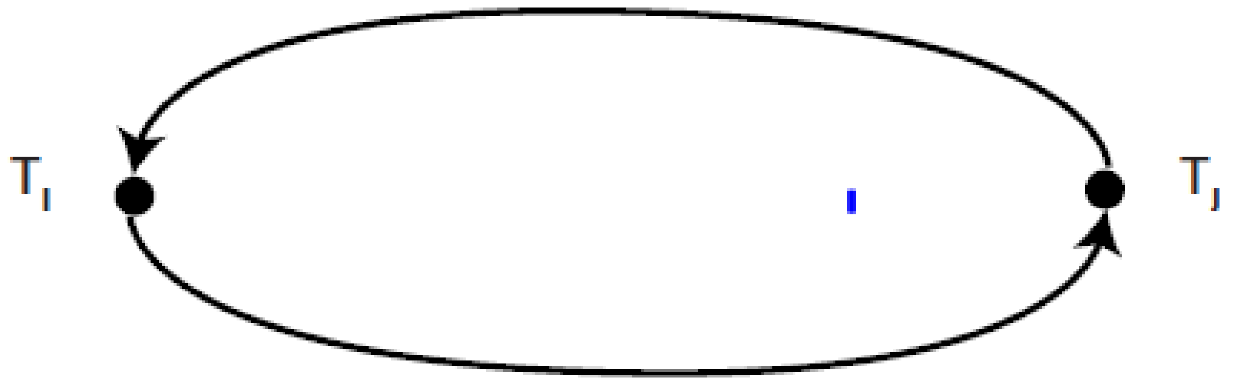
Phases of Optimistic Transaction Execution

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.
- Distributed DBMS requires special procedures to handle 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.
- A deadlock is a permanent phenomenon. If one exists in a system, it will not go away unless outside intervention takes place.(Usually from the user, the system operator, or the software system (the operating system or the distributed DBMS)).

Deadlock Management

- 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 Ti →Tj exists in the WFG if transaction Ti is waiting for Tj to release a lock on some entity.

Deadlock Management

- A deadlock occurs when the WFG contains a cycle.
- 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.(global deadlock)
- In distributed systems, 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.

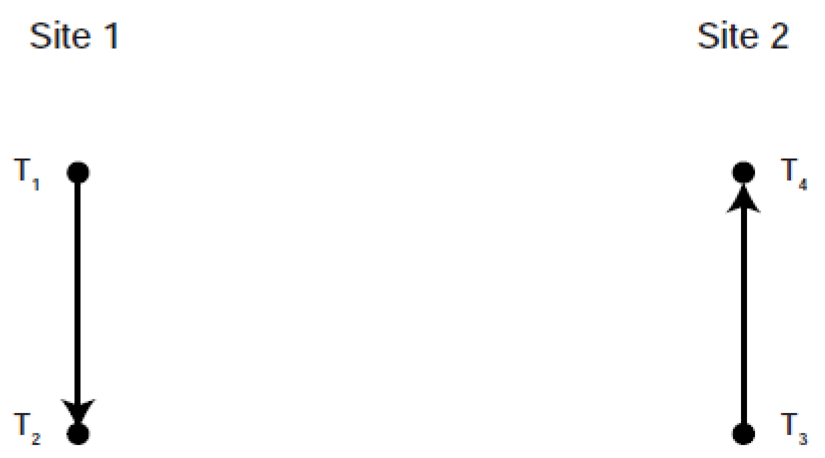
Deadlock Management

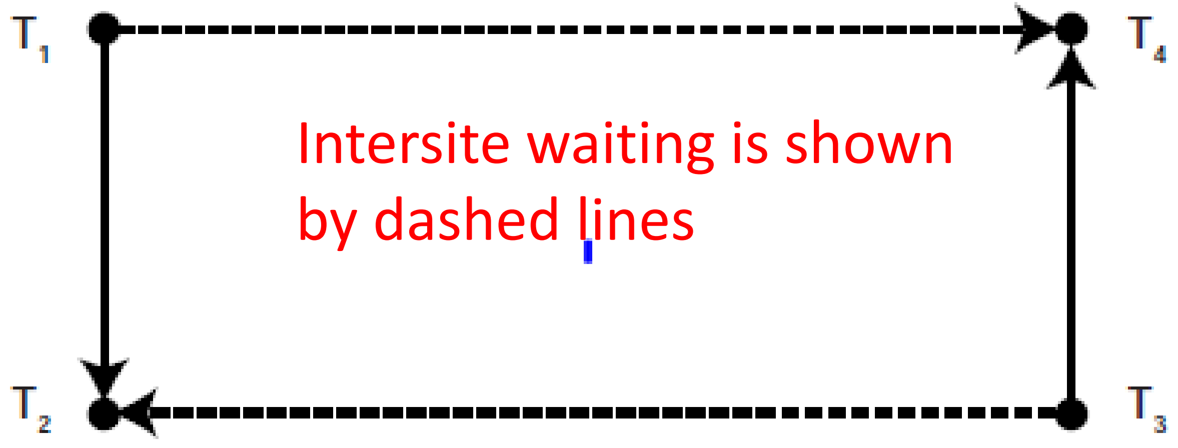
- Consider four transactions T1;T2;T3, and T4 with the following wait-for relationship among them:

$$T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_1$$

- If T1 and T2 run at site 1 while T3 and T4 run at site 2, 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, by examining the GWFG

Deadlock Management





Three known methods for handling deadlocks exist:

Prevention,Avoidance, detection and resolution.

Deadlock Prevention

- Deadlock prevention methods guarantee that deadlocks cannot occur in the first place.
- 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.
- The transaction manager reserves all the data items that are predeclared by a transaction that it allows to proceed.

Deadlock Prevention

- **Deadlock prevention methods** are not very suitable for database environments
  1. It is usually difficult to know precisely which data items will be accessed by a transaction.
  2. Access to certain data items may depend on conditions that may not be resolved until run time.
- To be safe, the system needs to consider the maximum set of data items, even if they end up not being accessed.
- This reduces concurrency and there is no need to abort and restart a transaction due to deadlocks.

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.
  - ➢Ordering the resources and insist that each process, request access to these resources, in that order.(proposed for operating systems)
  - ➢For Distributed database systems it is the lock units that are ordered and transactions always request locks in that order
  - ➢The ordering of lock units may be done either globally or locally at each site
  - ➢It is also necessary to order the sites and require that transactions which access data items at multiple sites request their locks by visiting the sites in the predefined order.

Deadlock Avoidance

- Another alternative is to make use of transaction timestamps to prioritize transactions and resolve deadlocks by aborting transactions with higher (or lower) priorities
- If a lock request of a transaction Ti is denied, the lock manager does not automatically force Ti to wait. Instead, it applies a prevention test to both the requesting transaction and the transaction that currently holds the lock (say Tj).
- If the test is passed, Ti is permitted to wait for Tj ; otherwise, one transaction or the other is aborted.

Timestamp algorithms

- **WAIT-DIE** and **WOUND-WAIT** algorithms are specific examples
- **WAIT-DIE Rule**. If T1 requests a lock on a data item that is already locked by T2, T1 is permitted to wait iff T1 is older than T2. If T1 is younger than T2, then T1 is aborted and restarted with the same timestamp.
- **WOUND-WAIT Rule.** If T1 requests a lock on a data item that is already locked by T2, then T1 is permitted to wait if and only if it is younger than T2; otherwise, T2 is aborted and the lock is granted to T1.

Algorithms

Deadlock avoidance methods are more suitable than prevention schemes for database environments.
Their fundamental drawback is that they require run-time support for deadlock management, which adds to the run-time overhead of transaction execution.

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

Deadlock Detection and Resolution

- Deadlock detection and resolution is the most popular 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 pre-empted 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,

Factors to be considered before abort

1. The amount of effort that has already been invested in the transaction.
2. The cost of aborting the transaction. This cost 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. 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).

QUESTIONS

REVISION QUESTIONS

- What are the functions of local recovery managers (LRM)?
- Describe the theory of serializability in distributed transactions.
- Describe how the Two-Phase Locking (2PL) works
- How does the sequence of execution of operations of a transaction differ between pessimistic and optimistic concurrency?
- How can timestamps be assigned in a multi-site transaction?
- Compare the WAIT-DIE and WOUND-WAIT algorithms

QUESTIONS

1. Which of the following histories are conflict equivalent?
2. Which of the below histories are serializable?

$$H_1 = \{W_2(x), W_1(x), R_3(x), R_1(x), W_2(y), R_3(y), R_3(z), R_2(x)\}$$

$$H_2 = \{R_3(z), R_3(y), W_2(y), R_2(z), W_1(x), R_3(x), W_2(x), R_1(x)\}$$

$$H_3 = \{R_3(z), W_2(x), W_2(y), R_1(x), R_3(x), R_2(z), R_3(y), W_1(x)\}$$

$$H_4 = \{R_2(z), W_2(x), W_2(y), W_1(x), R_1(x), R_3(x), R_3(z), R_3(y)\}$$

3. Give the algorithms for the transaction managers and the lock managers for the distributed two-phase locking approach.