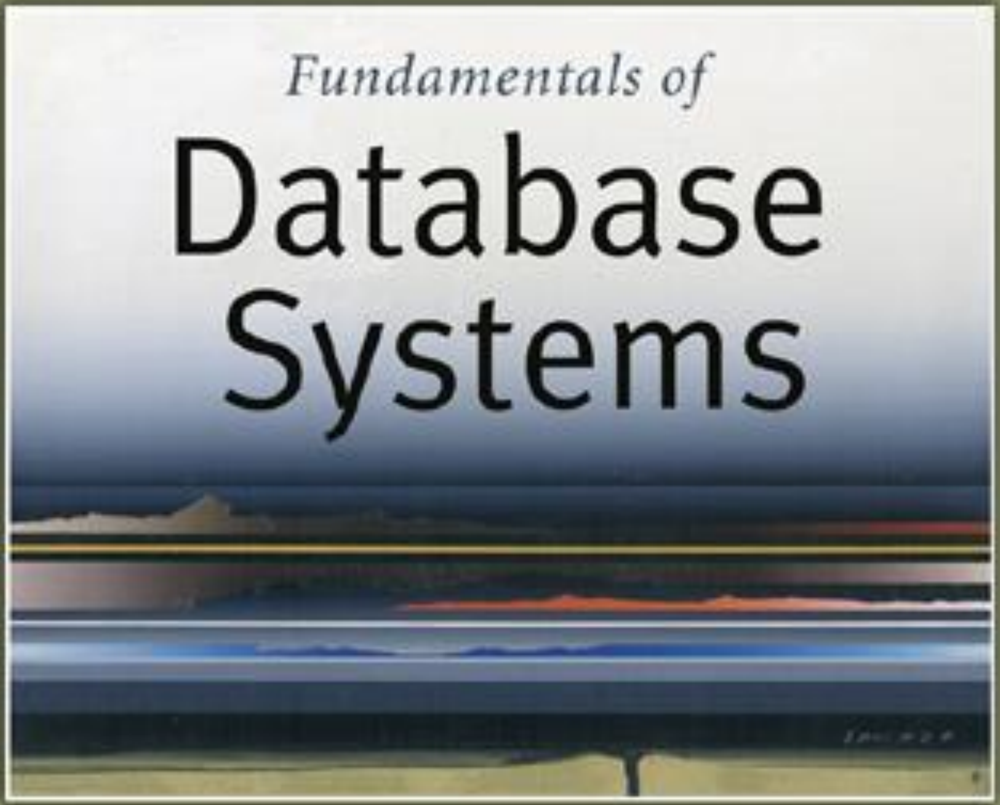


*Fundamentals of*

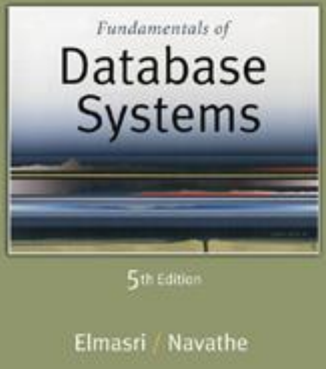
# Database Systems

5th Edition

Elmasri / Navathe

# Chapter 18

## Concurrency Control Techniques


Fundamentals of
Database Systems
5th Edition
Elmasri / Navathe

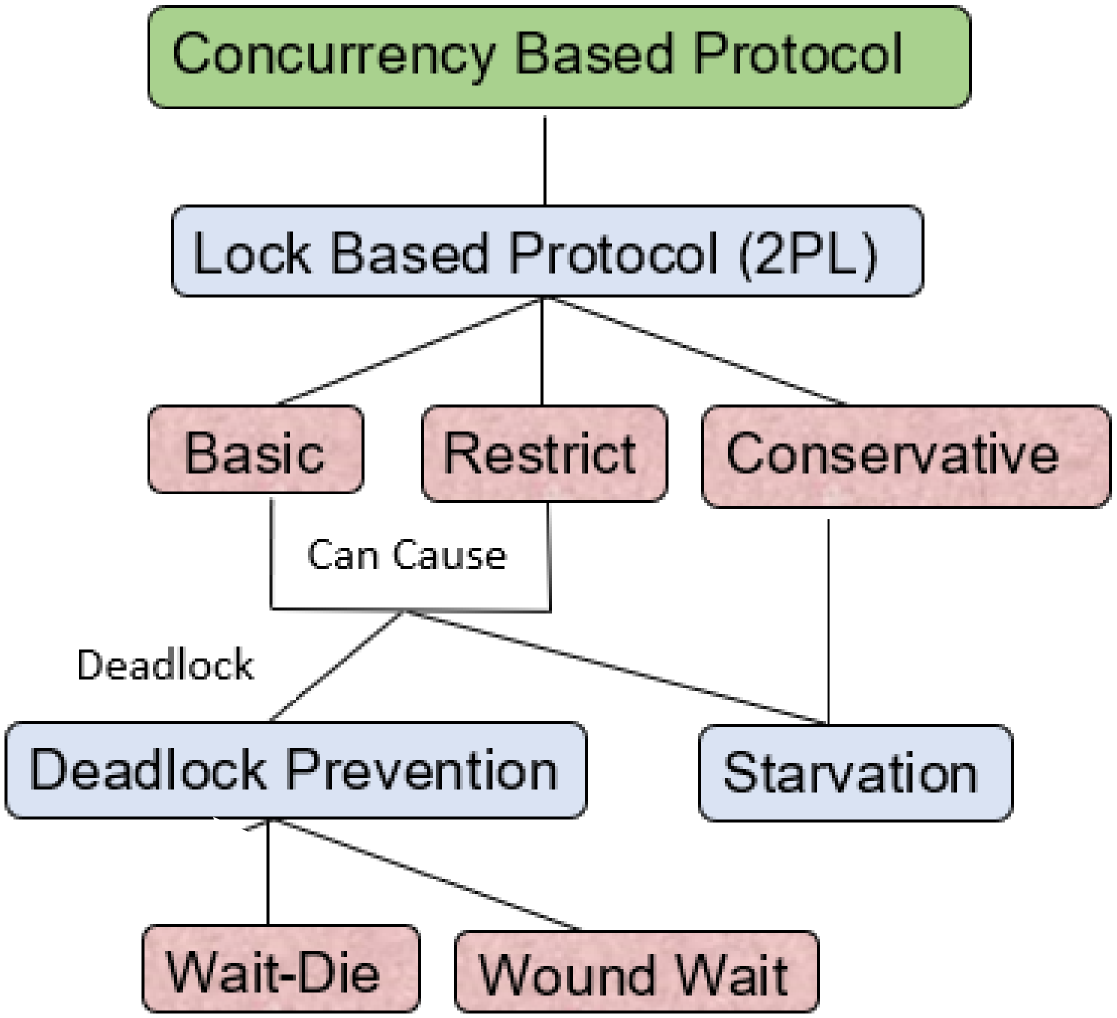
# Chapter 18 Outline

- Databases Concurrency Control
  1. Purpose of Concurrency Control
  2. Two-Phase locking
  3. Limitations of CCMs
  4. Index Locking
  5. Lock Compatibility Matrix
  6. Lock Granularity

## Database Concurrency Control

- 1 Purpose of Concurrency Control
  - To enforce Isolation (through mutual exclusion) among conflicting transactions.
  - To preserve database consistency through consistency preserving execution of transactions.
  - To resolve read-write and write-write conflicts.
- Example:
  - In concurrent execution environment if T1 conflicts with T2 over a data item A, then the existing concurrency control decides if T1 or T2 should get the A and if the other transaction is rolled-back or waits.

## Database Concurrency Control



## Database Concurrency Control

### Two-Phase Locking Techniques

- Locking is an operation which secures
  - (a) permission to Read
  - (b) permission to Write a data item for a transaction.
- Example:
  - Lock (X). Data item X is locked on behalf of the requesting transaction.
- Unlocking is an operation which removes these permissions from the data item.
- Example:
  - Unlock (X): Data item X is made available to all other transactions.
- Lock and Unlock are Atomic operations.

# Database Concurrency Control

## Two-Phase Locking Techniques: **Essential components**

- Two locks modes:
  - (a) shared (read) (b) exclusive (write).
- Shared mode: shared lock (X)
  - More than one transaction can apply share lock on X for reading its value, but NO WRITE LOCK can be applied on X by any other transaction.
- Exclusive mode: Write lock (X)
  - Only ONE write lock on X can exist at any time and NO SHARED LOCK can be applied by any other transaction on X.
  - Conflict matrix

| | Read | Write |
|---|---|---|
| Read | Y | N |
| Write | N | N |

## Database Concurrency Control

Two-Phase Locking Techniques: **Essential components**

- Lock Manager:
  - Managing locks on data items.
- Lock table:
  - Lock manager uses it to store the identity of transaction locking a data item, the data item, lock mode and pointer to the next data item locked. One simple way to implement a lock table is through linked list.

| Transaction ID | Data item id | lock mode | Ptr to next data item |
|---|---|---|---|
| T1 | X1 | Read | Next |

# Database Concurrency Control

Two-Phase Locking Techniques: **Essential components**

- Database requires that all transactions should be well-formed. A transaction is well-formed if:
  - It must lock the data item before it reads or writes to it.
  - It must not lock an already locked data items and it must not try to unlock a free data item.

# Database Concurrency Control

## Two-Phase Locking Techniques: Essential components

- The following code performs the **lock** operation:

```
B:if LOCK (X) = 0 (*item is unlocked*)
     then LOCK (X) ← 1 (*lock the item*)
   else (*item is locked*)
    begin
     wait (until lock (X) = 0) and
     the lock manager wakes up the transaction);
   goto B
   end;
```

# Database Concurrency Control

## Two-Phase Locking Techniques: Essential components

- The following code performs the **unlock** operation:

```
LOCK (X) ← 0 (*unlock the item*)
if any transactions are waiting then
  wake up one of the waiting the transactions;
```

# Database Concurrency Control

## Two-Phase Locking Techniques: Essential components

- The following code performs the **read/write** operation:

```
if LOCK (X) =0  (*unlocked*)    then
    begin
     LOCK (X) ← “read-locked”;
     no_of_reads (X) ← 1;
    end
 else if LOCK (X) ← “read-locked” (*locked)   then
        no_of_reads (X) ← no_of_reads (X) +1
 else
      begin
        wait (until LOCK (X) = “unlocked” and the lock manager wakes up the
        transaction);
        go to B
      end;
```

# Database Concurrency Control

## Two-Phase Locking Techniques: Essential components

- The following code performs the **unlock** operation:

```
if LOCK (X) = "write-locked"   then
 begin
     LOCK (X) ← "unlocked";
     wakes up one of the transactions, if any
 end
 else if LOCK (X) ← "read-locked" then
     begin
          no_of_reads (X) ← no_of_reads (X) -1
          if  no_of_reads (X) = 0 then
          begin
          LOCK (X) = "unlocked";
          wake up one of the transactions, if any
          end
     end;
```

# Database Concurrency Control

## Two-Phase Locking Techniques: **Essential components**

- Lock conversion
  - Lock upgrade: existing read lock to write lock

```
if Ti has a read-lock (X) and Tj has no read-lock (X) (i ≠ j) then
 convert read-lock (X) to write-lock (X)
else
 force Ti to wait until Tj unlocks X
```

  - Lock downgrade: existing write lock to read lock

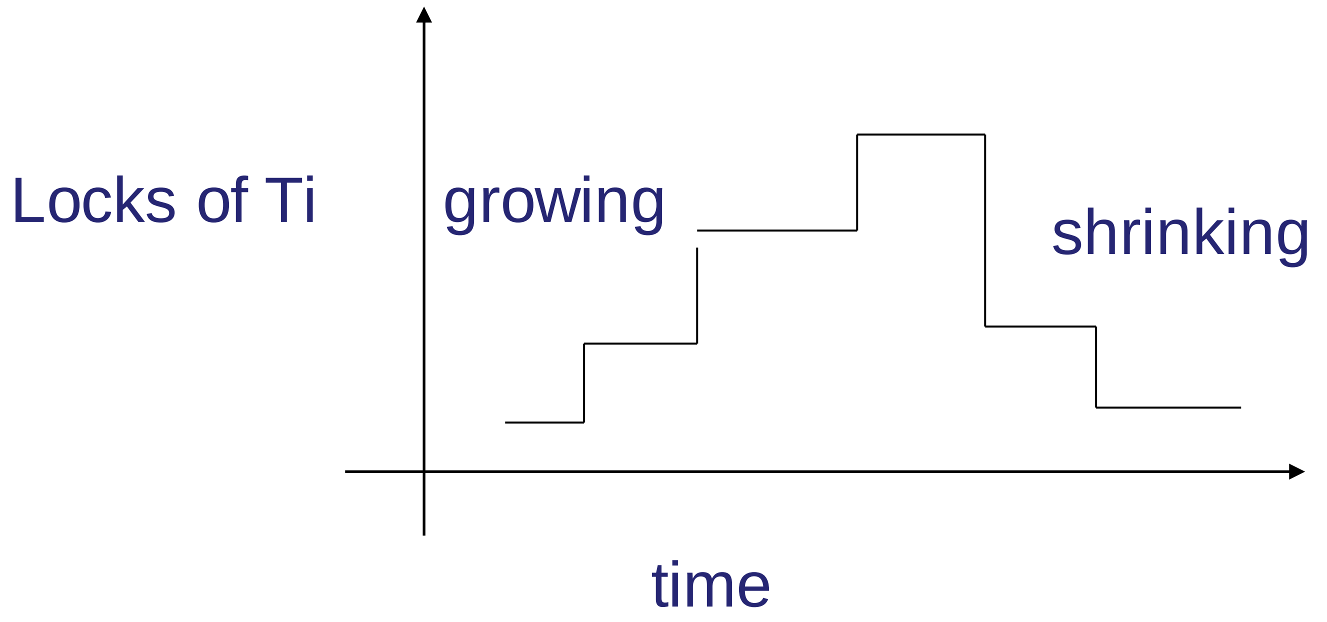
```
Ti has a write-lock (X)    (*no transaction can have any lock on X*)
convert write-lock (X) to read-lock (X)
```

## Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

- Two Phases:
  - (a) Locking (Growing)
  - (b) Unlocking (Shrinking)
- **Locking (Growing) Phase:**
  - A transaction applies locks (read or write) on desired data items one at a time (for each item one item then the other item)
- **Unlocking (Shrinking) Phase:**
  - A transaction unlocks its locked data items one at a time (when I finish item it unlock it then same for the other items)
- **Requirement:**
  - For a transaction, these two phases must be mutually exclusively, that is, <u>during locking phase unlocking phase must not start </u>and <u>during unlocking phase locking phase must not begin</u>.(when we are locking no unlocking is allowed, and vise versa)

## Two phase locking



Each transaction has to follow, no locking anymore after the first unlocking

# Database Concurrency Control

## Two-Phase Locking Techniques: The algorithm

| | T1 | T2 | |
|---|---|---|---|
| | **read_lock (Y);** | | |
| 50 | read_item (Y); | | |
| | **unlock (Y);** | | |
| | | **read_lock (X);** | |
| | | read_item (X); | 50 |
| Time | | **unlock (X);** | |
| | | **write_lock (Y);** | |
| | | read_item (Y); | |
| | | Y:=X+Y; | 100 |
| | | write_item (Y); | |
| | | **unlock (Y);** | |
| | **write_lock (X);** | | |
| | read_item (X); | | |
| 100 | X:=X+Y; | | |
| | write_item (X); | | |
| | **unlock (X);** | | |

**Result**

**X=50; Y=50**
**Non-serializable because it.**
**violated two-phase policy.**

## Cont.

- Transactions T1 and T2 of the previous figure do not follow the two-phase locking protocol because the write_lock(X) operation follows the unlock(Y) operation in T1, and similarly the write_lock(Y) operation follows the unlock(X) operation in T2.

- If we enforce two-phase locking , the transactions can be rewritten as T'1 and T'2 in next slide

# Database Concurrency Control

## Two-Phase Locking Techniques: The algorithm

```
T'1                    T'2

read_lock (Y);         read_lock (X);
read_item (Y);         read_item (X);
write_lock (X);        Write_lock (Y);
unlock (Y);            unlock (X);
read_item (X);         read_item (Y);
X:=X+Y;                Y:=X+Y;
write_item (X);        write_item (Y);
unlock (X);            unlock (Y);
```

T1 and T2 follow two-phase policy but they are subject to deadlock, which must be dealt with.

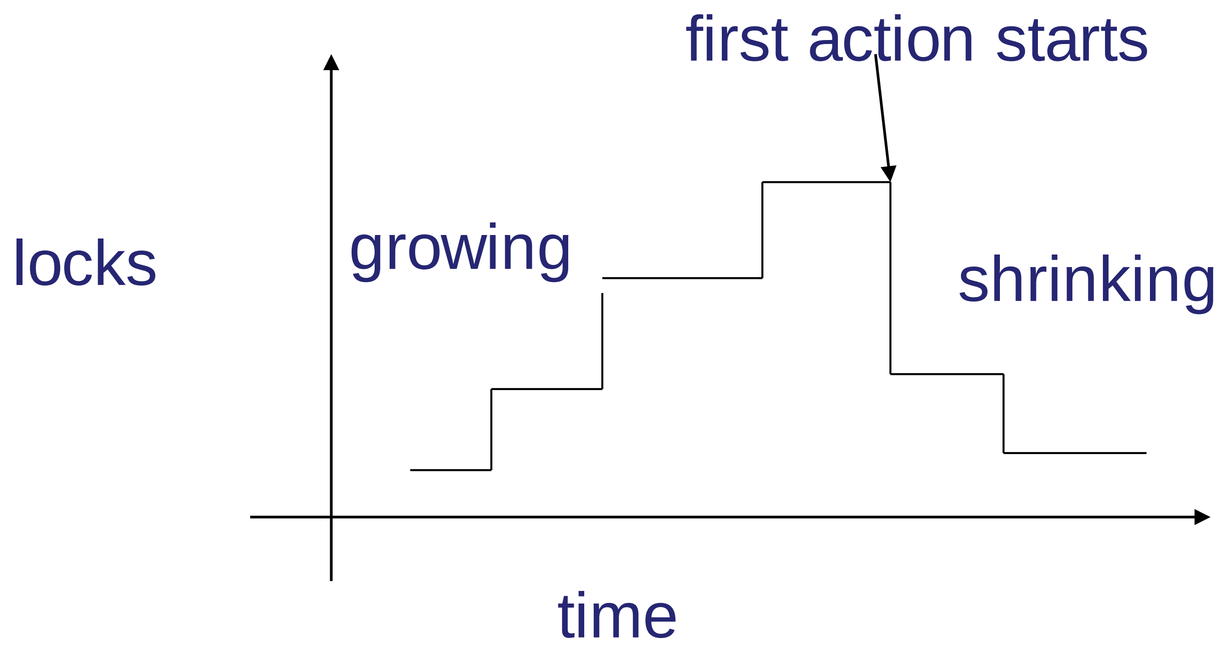
# Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

- Two-phase policy generates two locking algorithms
  - (a) **Basic**
  - (b) **Conservative**
- **Basic**:
  - Transaction locks data items incrementally. This may cause deadlock which is dealt with.
- **Conservative**:
  - Prevents deadlock by locking all desired data items before transaction begins execution.
- **Strict**:
  - A stricter version of Basic algorithm where unlocking is performed after a transaction terminates (commits or aborts and rolled-back). This is the most used two-phase locking algorithm.
  - Is not deadlock-free

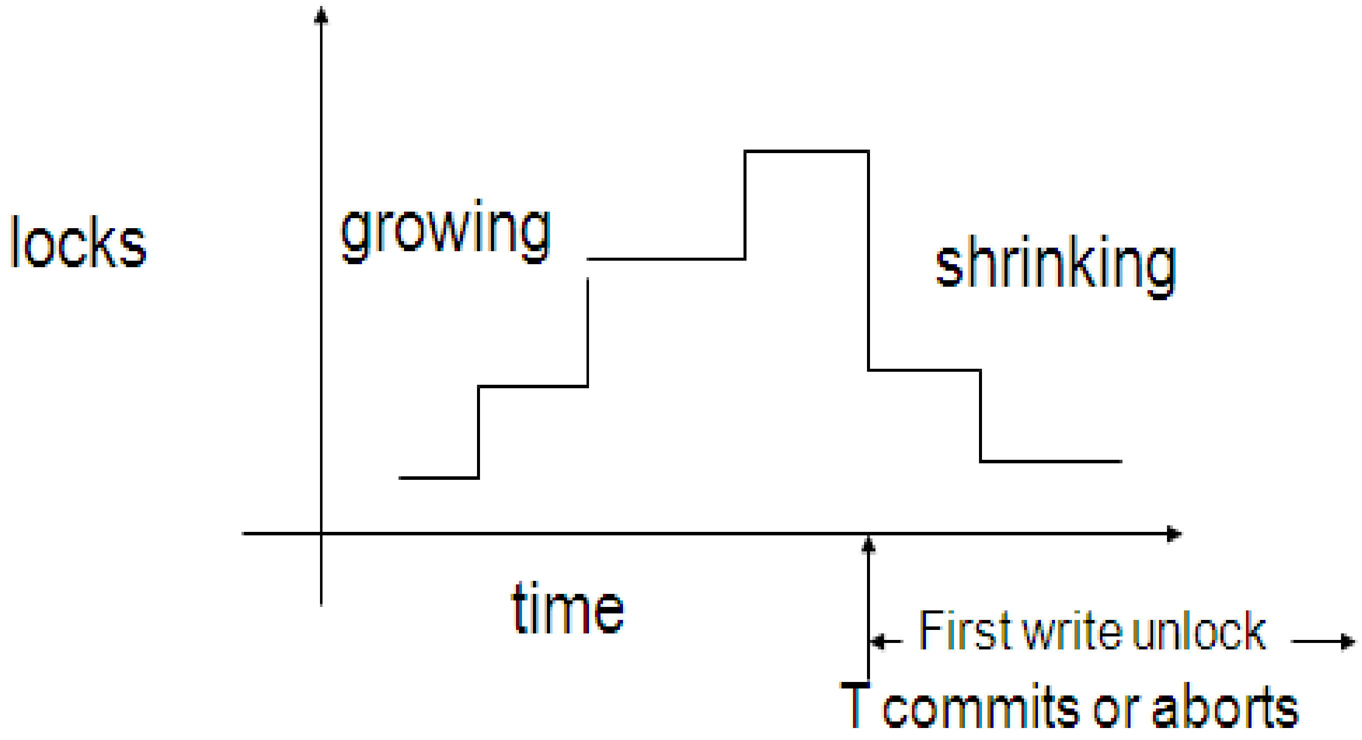
## Conservative 2PL

- Difficult but deadlock free



# Strict 2PL

- T does not release any write locks until it commits or aborts



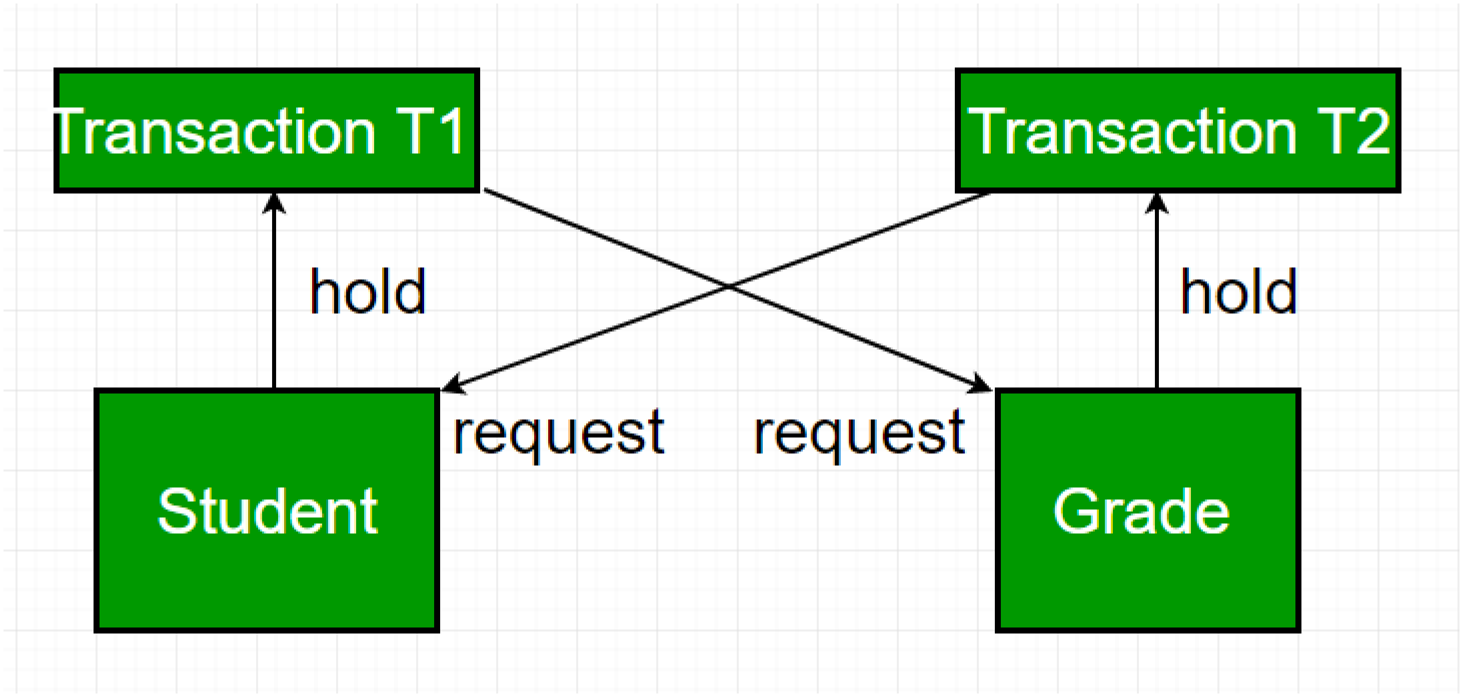
## Database Concurrency Control

**Deadlock**

- Suppose, Transaction T1 holds a lock on some rows in the Students table and needs to update some rows in the Grades table. Simultaneously,
- Transaction T2 holds locks on those very rows (Which T1 needs to update) in the Grades table but needs to update the rows in the Student table held by Transaction T1.
- Now, the main problem arises. Transaction T1 will wait for transaction T2 to give up the lock, and similarly, transaction T2 will wait for transaction T1 to give up the lock.
- Consequently, All activity comes to a halt and remains at a standstill forever unless the DBMS detects the deadlock and aborts one of the transactions.

# Database Concurrency Control

**Deadlock**



# Database Concurrency Control

## Dealing with Deadlock and Starvation

- **Deadlock**

| **T'1** | **T'2** |
|---|---|
| read_lock (Y); | |
| read_item (Y); | |
| | read_lock (X); |
| | read_item (X); |
| write_lock (X); | |
| (waits for X) | write_lock (Y); |
| | (waits for Y) |

T1 and T2 did follow two-phase policy but they are deadlock

- Deadlock (T'1 and T'2)

## Database Concurrency Control

### Dealing with Deadlock and Starvation

- **Deadlock prevention**
  - A transaction locks all data items it refers to before it begins execution.
  - This way of locking prevents deadlock since a transaction never waits for a data item.
  - The **conservative two-phase** locking uses this approach.

# Database Concurrency Control

## Dealing with Deadlock and Starvation

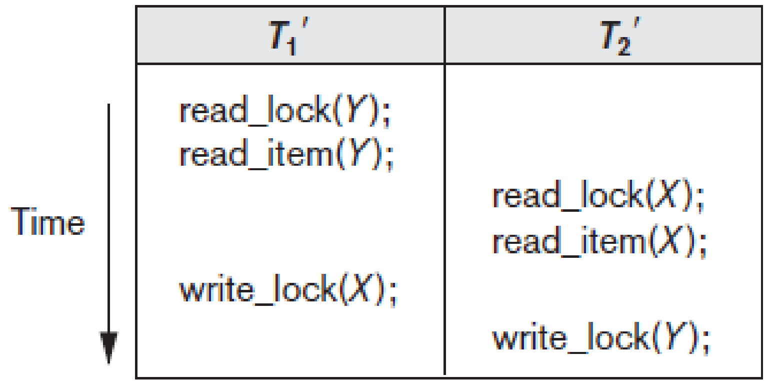
- **Deadlock detection and resolution**
  - In this approach, deadlocks are allowed to happen. The scheduler maintains a wait-for-graph for detecting cycle. If a cycle exists, then one transaction involved in the cycle is selected (victim) and rolled-back.
  - A wait-for-graph is created using the lock table. As soon as a transaction is blocked, it is added to the graph. When a chain like: Ti waits for Tj waits for Tk waits for Ti or Tj occurs, then this creates a cycle.

# Dealing with Deadlock (cont.)

**Deadlock Detection**

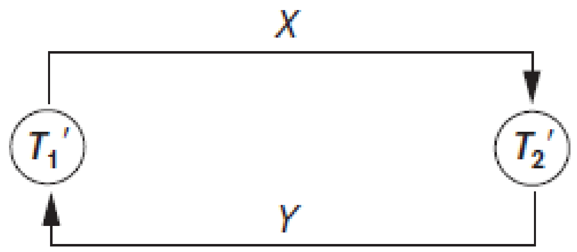
- Deadlock detection is usually handled by the construction of a wait-for graph (WFG) that shows the transaction dependencies
- We have a state of deadlock if and only if the wait-for graph has a cycle
- Kill one transaction which is called **victim selection**

(a)



| $T_1'$ | $T_2'$ |
|---|---|
| read_lock($Y$);<br>read_item($Y$); | |
| | read_lock($X$);<br>read_item($X$); |
| write_lock($X$); | |
| | write_lock($Y$); |

(b)



$$\begin{tikzcd} T_1' \arrow[r, bend left, "X"] & T_2' \arrow[l, bend left, "Y"] \end{tikzcd}$$

## Dealing with Deadlock

### Deadlock Prevention

#### Timeouts

- Transaction that requests a lock will wait for only a system-defined period.
- If the lock has not been granted within this period, the lock request times out.
- In this case, the DBMS assumes the transaction may be deadlocked, even though it may not be, and it aborts and automatically restarts the transaction.
- This is a very simple and practical solution to deadlock prevention and is used by several commercial DBMSs.

## Concurrency Control based on Timestamp ordering

- **Timestamp**
  - is a unique identifier created by the DBMS to identify a transaction.
  - Typically, timestamp values are assigned in the order in which the transactions are submitted to the system, so a timestamp can be thought of as the transaction start time.
  - We will refer to the timestamp of transaction T as TS(T).
  - Concurrency control techniques based on timestamp ordering do not use locks; hence, deadlocks cannot occur.

## Database Concurrency Control

Dealing with Deadlock and Starvation

- **Deadlock avoidance**
  - There are many variations of two-phase locking algorithm.
  - Some avoid deadlock by not letting the cycle to complete.
  - That is as soon as the algorithm discovers that blocking a transaction is likely to create a cycle, it rolls back the transaction.
  - Wound-Wait and Wait-Die algorithms use **timestamps** to avoid deadlocks by rolling-back victim.

# Database Concurrency Control

Timestamp based concurrency control algorithm

- **Timestamp**
  - A monotonically increasing variable (integer) indicating the age of an operation or a transaction. A larger timestamp value indicates a more recent event or operation.
  - Timestamp based algorithm uses timestamp to serialize the execution of concurrent transactions.

## Cont.

Timestamps can be **generated** in several ways:

- One possibility is to use a counter that is incremented each time its value is assigned to a transaction. The transaction timestamps are numbered 1, 2, 3, ... in this scheme..
- Another way to implement timestamps is to use the current date/time value of the system clock and ensure that no two timestamp values are generated during the same tick of the clock.

## Database Concurrency Control

- The basic timestamp ordering protocol (BASIC T/O) allows reads and writes on database objects without using locks.
- Each active database item will have the following two timestamps:

  read_TS(X): the number of the youngest transaction that read item X.

  write_TS(X): the number of the youngest transaction that wrote item X.
- Every database object X is tagged with timestamp of the last transaction that successfully performed a read or write on that object.

# Database Concurrency Control

Example:

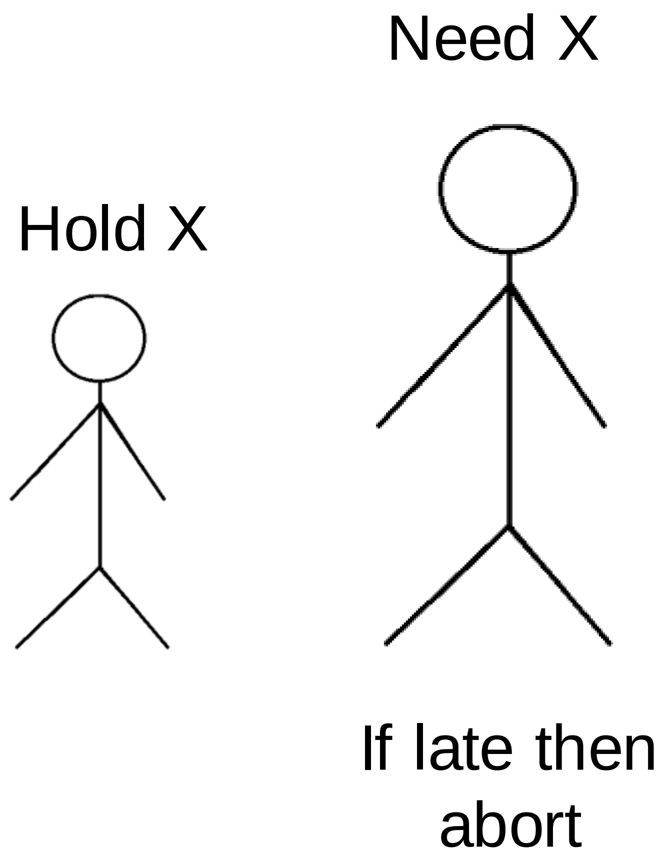
- Consider two transactions T5 and T8.
- Where T8 has started after T5.
- Hence, T8 is **younger** to T5.
- Also, TS(T8) > TS(T5).
- T8→ Is the younger but the one who came after T5 so have larger TS.

## Database Concurrency Control

Timestamp based concurrency control algorithm

- **Basic Timestamp Ordering**
  1. Transaction T issues a write_item(X) operation:
     - If read_TS(X) > TS(T) or if write_TS(X) > TS(T), then an younger transaction (new transaction entered after you) has already read/write the data item so abort and roll-back T and reject the operation.
     - If the condition in part (a) does not exist, then execute write_item(X) of T and set write_TS(X) to TS(T).
  2. Transaction T issues a read_item(X) operation:
     - If write_TS(X) > TS(T), then an younger transaction has already written to the data item so abort and roll-back T and reject the operation.
     - If write_TS(X) ≤ TS(T), then execute read_item(X) of T and set read_TS(X) to the larger of TS(T) and the current read_TS(X).

## Database Concurrency Control



## Dealing with Deadlock (cont.)

**Deadlock Prevention**

- Use the concept of transaction timestamp.

- Timestamp is unique identifier is typically based on the order in which transactions are started

- Notice that the older transaction (which starts first) has the smaller timestamp value.

- 2 techniques are used: **Wait-Die and Wound-Wait**

## Dealing with Deadlock (cont.)

**Deadlock Prevention**

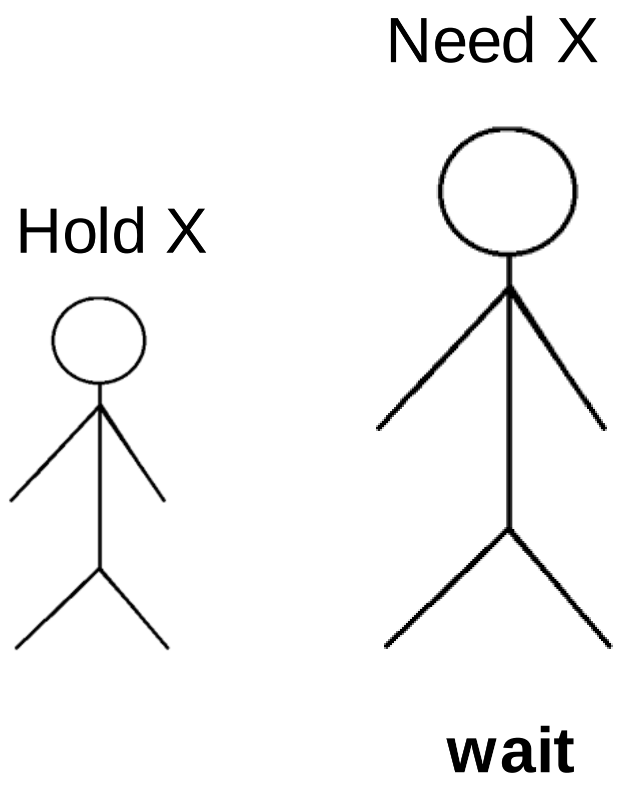
**1. Wait-Die Technique**

- If the older transaction request item held by the younger one, then the older will wait for the item to be unlocked.
- If the younger transaction requests item held by the older one, then the younger will be aborted and rollback

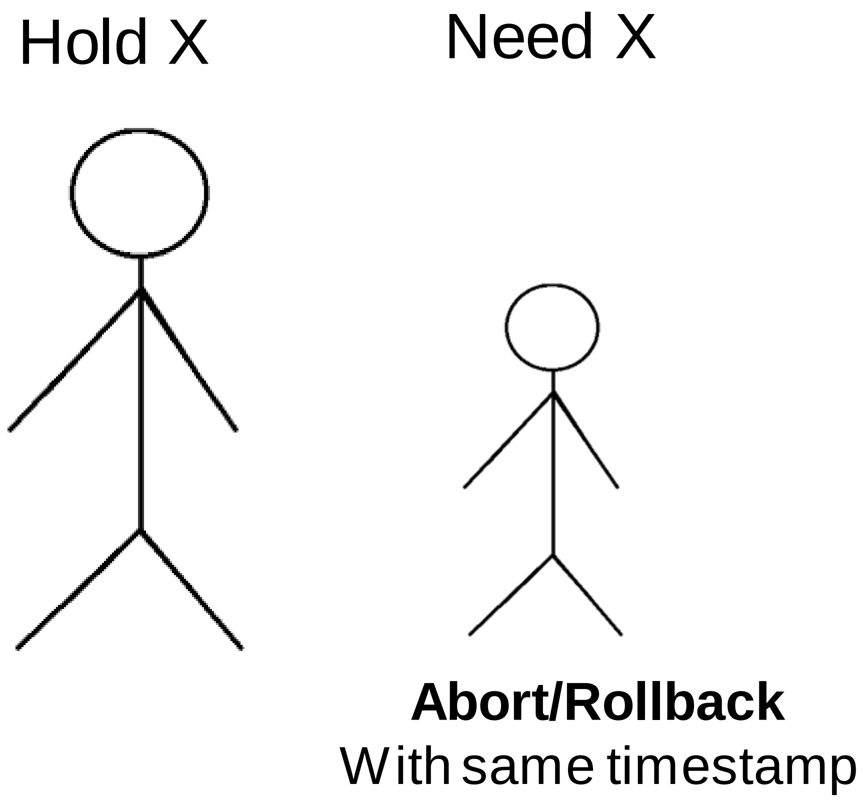
If $TS(T_i) < TS(T_j)$, then ($T_i$ older than $T_j$) $T_i$ is allowed to wait; otherwise ($T_i$ younger than $T_j$) abort $T_i$ ($T_i$ dies) and restart it later with the same timestamp

## Dealing with Deadlock (cont.)
## Wait-Die Technique

- 1st Scenario



- 2nd Scenario



# Dealing with Deadlock (cont.)

**Deadlock Prevention**

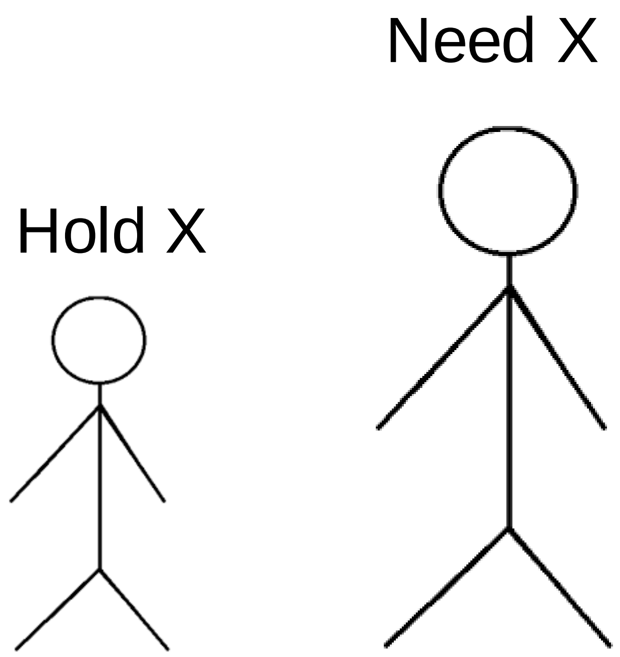
**2. Wound-Wait Technique** (opposite of Wait-Die)

- If the older transaction request item held by the younger one, then the younger one will die.
- If the younger transaction requests item held by the older one, then the younger will be wait

If $TS(T_i) < TS(T_j)$, then ($T_i$ older than $T_j$) abort $T_j$ ($T_i$ wounds $T_j$) and restart it later with the same timestamp; otherwise ($T_i$ younger than $T_j$) Ti is allowed to wait

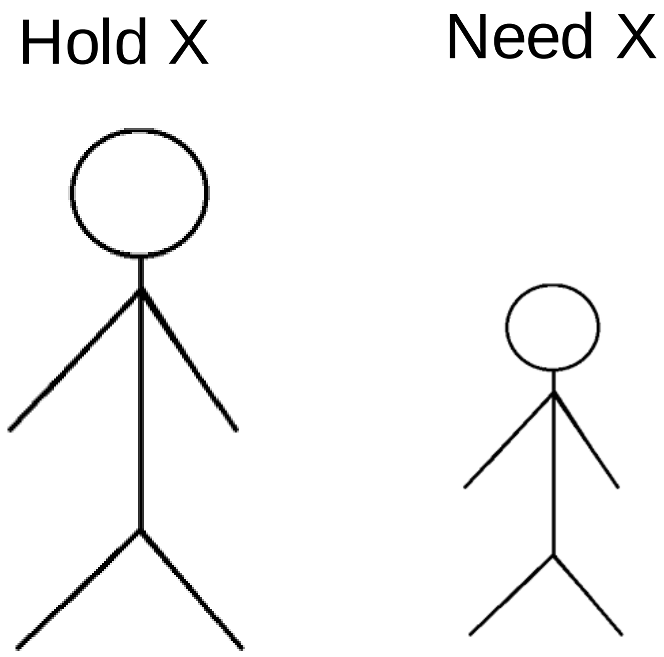
## Dealing with Deadlock (cont.)
## Wound-Wait Technique

- 1st Scenario



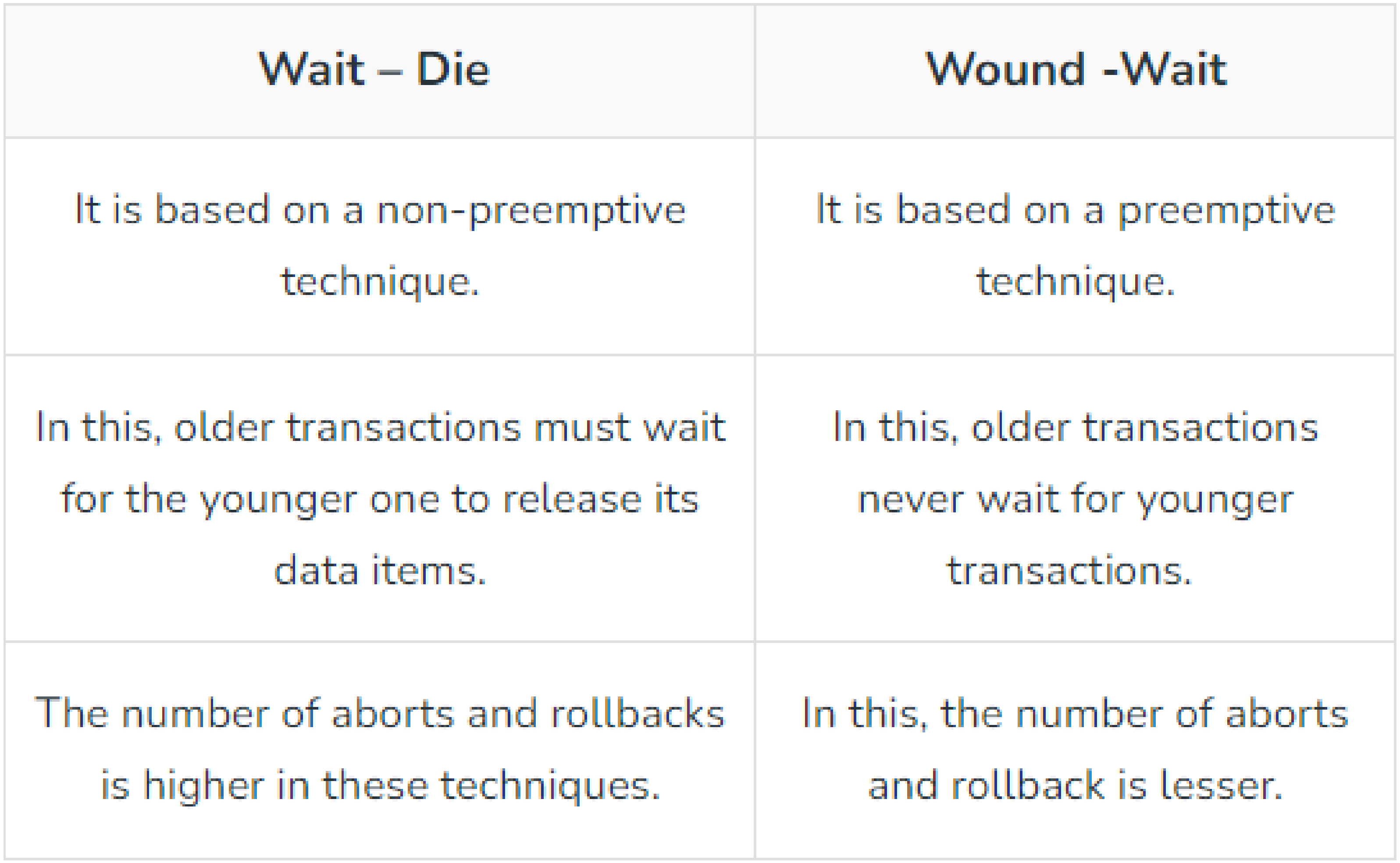
**Die**
& restart with same time stamp

- 2nd Scenario



**Wait**

## Dealing with Deadlock (cont.)



| Wait – Die | Wound -Wait |
|---|---|
| It is based on a non-preemptive technique. | It is based on a preemptive technique. |
| In this, older transactions must wait for the younger one to release its data items. | In this, older transactions never wait for younger transactions. |
| The number of aborts and rollbacks is higher in these techniques. | In this, the number of aborts and rollback is lesser. |

## Dealing with Deadlock (cont.)

- Thus, both schemes end up aborting the younger of the two transactions that may be involved in a deadlock.
- It is done based on the assumption that aborting the younger transaction will waste less processing which is logical.
- In such a case there cannot be a cycle since we are waiting linearly in both cases.

## Conclusion

- In conclusion, timestamp-based concurrency control and deadlock prevention schemes are important techniques in database management systems to ensure transaction correctness, consistency, and concurrency. These techniques help maintain data integrity and prevent situations where transactions are deadlocked or rolled back, ensuring efficient and effective database operations.

# Database Concurrency Control

## Dealing with Deadlock and Starvation

- **Starvation**
  - Starvation occurs when a particular transaction consistently waits or restarted and never gets a chance to proceed further.
  - In a deadlock resolution it is possible that the same transaction may consistently be selected as victim and rolled-back.
  - This limitation is inherent in all priority-based scheduling mechanisms.
  - In Wound-Wait scheme a younger transaction may always be wounded (aborted) by a long running older transaction which may create starvation.
  - The most common solution is to include the number of rollbacks in the cost factor or First-come-first-served.

Questions?