Concurrency Control and Recovery Techniques

- Lock-based
- Timestamp-based
- Log based Recovery
- Deadlock handling

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

1. Two-Phase Locking Techniques(2PL)

–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.

Locking Techniques for Concurrency Control

A **lock**: a variable associated with a data item that describes the status of the item with respect to possible operations that can be applied to it. Generally, there is one lock for each data item in the database.

(**Granularity of locking** varies : typically rows or sets of rows. An entire relation may be locked, or an entire database.)

TYPES OF LOCKS:

• **Binary locks:** only two states of a lock; too simple and too restrictive; not used in practice.

• **Shared/Exclusive locks:** which provide more general locking capabilities and are used in practical database locking schemes. (Read Lock as a shared lock, Write Lock as an exclusive lock).

• **Conversion of locks:** used to improve performance of locking protocols.

Binary locks

- A **binary lock** can have two **states** or **values**: locked and unlocked (or 1 and 0, for simplicity).

- A binary lock enforces **mutual exclusion** on the data item; i.e., at a time only one transaction can hold a lock.

- A distinct lock is associated with each database item X. If the value of the lock on X is 1, item X *cannot be accessed* by a database operation that requests the item. If the value of the lock on X is 0, the item can be accessed when requested.

Operations on Binary Locks:

- **lock_item (X) :**
  A transaction requests access to an item X by first issuing a **lock_item(X)** operation. If LOCK(X) = 1, the transaction is forced to wait. If LOCK(X) = 0, it is set to 1 (the transaction **locks** the item) and the transaction is allowed to access item X.

- **unlock_item(X)** :
  Sets LOCK(X) to 0 (**unlocks** the item) so that X may be accessed by other transactions.

IMPLEMENTATION OF LOCK_ITEM:

- A binary-valued variable, LOCK, associated with each data item X in the database.

LOCK TABLE:

A table of rows with three fields:
<data item name, LOCK, locking transaction>

plus a queue for transactions that are waiting to access the item.
Table may have a hashed organization by data item name.

Lock manager :
a subsystem to keep track of and control access to locks.

BINARY LOCKING SCHEME

Every transaction must obey the following rules.
Rules are enforced by the **LOCK MANAGER**

1. A transaction T must issue the operation lock_item(X) before any read_item(X) or write_item(X) operations are performed in T.

2. A transaction T must issue the operation unlock_item(X) after all read_item(X) and write_item(X) operations are completed in T.

3. A transaction T will not issue a lock_item(X) operation if it already holds the lock on item X .

4. A transaction T will not issue an unlock_item(X) operation on X unless it already holds the lock on item X.

```
lock_item(X):
B:  if LOCK(X) = 0                    (* item is unlocked *)
        then LOCK(X) ← 1              (* lock the item *)
    else
        begin
        wait (until LOCK(X) = 0
            and the lock manager wakes up the transaction);
        go to B
        end;
unlock_item(X):
    LOCK(X) ← 0;                      (* unlock the item *)
    if any transactions are waiting
        then wakeup one of the waiting transactions;
```

Figure 22.1
Lock and unlock operations for binary locks.

Shared/Exclusive (or Read/Write) locks

A lock associated with an item X, LOCK(X), now has three possible states:
“read-locked,” “write-locked,” or “unlocked.”

A **read-locked item** is also called **share-locked**, because other transactions are allowed to read the item.

A **write-locked item** is called **exclusive-locked**, because a single transaction exclusively holds the lock on the item.

LOCK TABLE:

Lock table will have four fields:
<data item name, LOCK, no_of_reads, locking_transaction(s)>.
Value of LOCK : SOME ENCODED VALUE FOR READ /ENCODED VALUE FOR WRITE

- LOCK(X)=write-locked, the value of locking_transaction(s) is a single transaction that holds the exclusive (write) lock on X.

- LOCK(X)=read-locked, the value of locking transaction(s) is a list of one or more transactions that hold the shared (read) lock on X.

Shared/Exclusive (or Read/Write) locks Cont..

RULES FOR Read/Write LOCKS

1. **A transaction T must issue the operation read_lock(X) or write_lock(X) before any read_item(X) operation is performed in T.**

2. **A transaction T must issue the operation write_lock(X) before any write_item(X) operation is performed in T.**

3. **A transaction T must issue the operation unlock(X) after all read_item(X) and write_item(X) operations are completed in T.**

4. **A transaction T will not issue a read_lock(X) operation if it already holds a read (shared) lock or a write (exclusive) lock on item X.**

(EXCEPTIONS: DOWNGRADING OF LOCK from WRITE TO READ)

5. **A transaction T will not issue a write_lock(X) operation if it already holds a read (shared) lock or write (exclusive) lock on item X.**

(EXCEPTIONS: UPGRADING OF LOCK FROM READ TO WRITE)

6. **A transaction T will not issue an unlock(X) operation unless it already holds a read (shared) lock or a write (exclusive) lock on item X.**

```
read_lock(X):
B:  if LOCK(X) = "unlocked"
        then begin LOCK(X) ← "read-locked";
                no_of_reads(X) ← 1
                end
    else if LOCK(X) = "read-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;
write_lock(X):
B:  if LOCK(X) = "unlocked"
        then LOCK(X) ← "write-locked"
    else begin
            wait (until LOCK(X) = "unlocked"
                and the lock manager wakes up the transaction);
            go to B
            end;
unlock (X):
    if LOCK(X) = "write-locked"
        then begin LOCK(X) ← "unlocked";
                wakeup one of the waiting transactions, if any
                end
    else it 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";
                            wakeup one of the waiting transactions, if any
                            end
                end;
```

Figure 22.2
Locking and unlocking operations for two-mode (read-write or shared-exclusive) locks.

Conversion of Locks

UPGRADING:
If T *is the only transaction* holding a read lock on X at the time it issues the write_lock(X) operation, the lock can be **upgraded**; otherwise, the transaction must wait.

DOWNGRADING:
It is also possible for a transaction T to issue a write_lock(X) and then later on to **downgrade** the lock by issuing a read_lock(X) operation.

Guaranteeing Serializability by Two-phase Locking

A transaction is said to follow the **two-phase locking protocol** if *all* locking operations (read_lock, write_lock) precede the *first* unlock operation in the transaction.

TWO PHASES:

Expanding or **growing** (**first**) **phase**, during which new locks on items can be acquired but none can be released;

Shrinking (**second**) **phase**, during which existing locks can be released but no new locks can be acquired.

WITH LOCK CONVERSION:

Upgrading of locks (from read-locked to write-locked) must be done during the expanding phase,

Downgrading of locks (from write-locked to read-locked) must be done in the shrinking phase. Hence, a read_lock(X) operation that downgrades an already held write lock on X can appear only in the shrinking phase.

VARIATIONS OF 2-PHASE LOCKING

Basic, Conservative, Strict, and Rigorous :

1. BASIC Two-phase Locking
As described earlier.

2. CONSERVATIVE 2PL (OR **STATIC 2PL**)
Requires a transaction to lock all the items it accesses *before the transaction begins execution*, by **predeclaring** its *read-set* and *write-set*. .(the **read-set** of a transaction is the set of all items that the transaction reads, and the **write-set** is the set of all items that it writes.)

If any of the predeclared items needed cannot be locked, the transaction does not lock any item; instead, it waits until all the items are available for locking.

-POLICY - Lock all that you need before reading or writing. Transaction is in shrinking phase after it starts.

-PROPERTY: Conservative 2PL is a deadlock-free protocol

- PRACTICAL : Difficult to use because of difficulty in predeclaring the read-set and write-set.

STRICT 2PL:

the most popular variation of 2PL is **strict 2PL**, which guarantees strict schedules .In this variation, a transaction T does not release any of its exclusive (write) locks until after it commits or aborts. Hence, no other transaction can read or write an item that is written by T unless T has committed, leading to a strict schedule for recoverability.

- POLICY - Release write locks only after terminating. Transaction is in expanding phase until it ends (may release some read locks before commit).

-PROPERTY: NOT a deadlock-free protocol

- PRACTICAL : Possible to enforce and desirable due to recoverability.

<u>RIGOROUS 2PL:</u>

A transaction T does not release any of its locks (exclusive or shared) until after it commits or aborts Behaves similar to Strict 2 PL except it is more restrictive, but easier to implement since all locks are held till commit.

<u>LIMITATIONS OF 2 PL:</u>

Use of locks can cause two additional problems: deadlock and starvation.

Concurrency Control based on Time Stamp Ordering

- **Timestamp Ordering** of transactions is another way to produce serializable schedule without using locks.

Timestamp : a unique identifier created by the DBMS to identify a transaction. A timestamp can be thought of as the *transaction start time.*

- Timestamp of transaction T is referred to as **TS(T)**.
- **NO deadlocks** occur because there are no locks.

Timestamps generation:

1. The transaction timestamps are numbered 1, 2, 3, ... in this scheme. A computer counter has a finite maximum value, so the system must periodically reset the counter to zero.

2. Using the current value of the system clock and ensuring that no two timestamp values are generated during the same tick of the clock.

Timestamp ordering Algorithm

Basic Idea: to order the transactions based on their timestamps.

Why call it timestamp ordering (**TO**) **?**: the equivalent serial schedule has the transactions in order of their timestamp values.

TIMESTAMP ORDERING ALGORITHM:

The algorithm must ensure that, for each item accessed by *conflicting operations* in the schedule, the order in which the item is accessed does not violate the serializability order. To do this, the algorithm associates with each database item X two timestamp (**TS**) values:

1. **read_TS(X)**: The **read timestamp** of item X; this is the largest timestamp among all the timestamps of transactions that have successfully read item X (that is, read_TS(X) = TS(T), where T is the *youngest* (latest) transaction that has read X successfully).

2. **write_TS(X)**: The **write timestamp** of item X; this is the largest of all the timestamps of transactions that have successfully written item X (that is, write_TS(X) = TS(T), where T is the *youngest* (latest) transaction that has written X successfully).

BASIC TIMESTAMP ORDERING

Whenever some transaction T tries to issue a read_item(X) or a write_item(X) operation,
The **basic TO** algorithm compares the timestamp of T with the read timestamp: read_TS(X) and the write timestamp: write_TS(X) to ensure that the timestamp order of transaction execution is not violated.

RESULT OF TIMESTAMP ORDERING VIOLATION:

• If the timestamp order is violated, then transaction T is aborted and resubmitted to the system as a new transaction with a *new timestamp*.

• If T is aborted and rolled back, any transaction T1 that may have used a value written by T must also be rolled back. Similarly, any transaction T2 that may have used a value written by T1 must also be rolled back, and so on. This effect is known as **cascading rollback - a problem with the basic TO algorithm.**

• An *additional protocol* must be enforced to ensure that the schedules are recoverable, **cascadeless, or strict.**

Cascadeless schedule:

One where every transaction reads only the items that are written by committed transactions.

Strict Schedules:

A schedule in which a transaction can neither read or write an item X until the last transaction that wrote X has committed.

Basic TO algorithm

1. Transaction T issues a write_item(X) operation:

a) If read_TS(X) > TS(T) or if write_TS(X) > TS(T), then abort and roll back T and reject the operation. This should be done because some transaction with a timestamp greater than TS(T) and hence *after* T in the timestamp ordering has already read or written the value of item X before T had a chance to write X, thus violating the timestamp ordering.

(b) If the condition in part (a) does not occur, then execute the write_item(X) operation of T and set write_TS(X) to TS(T).

2. Transaction T issues a read_item(X) operation:

(a) If write_TS(X) > TS(T), then abort and roll back T and reject the operation. This should be done because some transaction with timestamp greater than TS(T) and hence *after* T in the timestamp ordering has already written the value of item X before T had a chance to read X.

(b) If write_TS(X) <= TS(T), then execute the read_item(X) operation of T and set read_TS(X) to the larger of TS(T) and the current read_TS(X).

Hence, the basic TO algorithm checks whenever two *conflicting operations* occur in the incorrect order, and rejects the later of the two operations by aborting the transaction that issued it.

VARIATIONS OF TIMESTAMP ORDERING

• deadlock does not occur with timestamp ordering. However, cyclic restart (and hence starvation) may occur if a transaction is continually aborted and restarted.

Strict TimeStamp Ordering

A variation of basic TO called **strict TO** ensures that the s**chedules are both strict (for easy recoverability) and (conflict) serializable.**

• a transaction T that issues a read_item(X) or write_item(X) such that TS(T) > write_TS(X) has its read or write operation *delayed* until the transaction T' that *wrote* the value of X (hence TS(T') = write_TS(X)) has committed or aborted.

To implement this algorithm, it is necessary to simulate the locking of an item X that has been written by transaction T' until T' is either committed or aborted. This algorithm does not cause deadlock, **since T waits for T' only if TS(T) > TS(T'), i.e., always wait on the older transaction to commit or abort.**

Thomas' write rule: A modification of the basic TO algorithm

A modification of the basic TO algorithm, known as **Thomas' write rule,** does not enforce conflict serializability; but it rejects fewer write operations, by *modifying the checks for the **write_item(X)** operation as follows*:

1. If read_TS(X) > TS(T), then abort and roll back T and reject the operation.

2. If write_TS(X) > TS(T), then do not execute the write operation but continue processing. THIS IS A REJECTED WRITE. This is because some transaction with timestamp greater than TS(T) and hence after T in the timestamp ordering has already written the value of X. Hence, we must ignore the write_item(X) operation of T because it is already outdated and obsolete. Notice that any conflict arising from this situation would be detected by case (a).

3. If neither the condition in part (a) nor the condition in part (b) occurs, then execute the write_item(X) operation of T and set write_TS(X) to TS(T).

Log Based Recovery

Log and log records –

- The log is a sequence of log records, **recording all the update activities in the database**.
- In a stable storage, logs for each transaction are maintained. Any operation which is performed on the database is recorded is on the log.
- Prior to performing any modification to database, an update log record is created to reflect that modification.

An update **log record represented as: <Ti, Xj, V1, V2>** has these fields:

1. **Transaction identifier:** Unique Identifier of the transaction that performed the write operation.
2. **Data item:** Unique identifier of the data item written.
3. **Old value:** Value of data item prior to write.
4. **New value:** Value of data item after write operation.

Other type of log records are:

1. **<Ti start>**: It contains information about when a transaction Ti starts.
2. **<Ti commit>**: It contains information about when a transaction Ti commits.
3. **<Ti abort>**: It contains information about when a transaction Ti aborts.

Log Based Recovery

Log record represented as: <Ti, Xj, V1, V2> - Example

Let's assume there is a transaction to modify the City of a student. The following logs are written for this transaction.

- When the transaction is initiated, then it writes 'start' log.

  <Tn, Start>

- When the transaction modifies the City from 'Noida' to 'Bangalore', then another log is written to the file.

  <Tn, City, 'Noida', 'Bangalore' >

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

  <Tn, Commit>

Log Based Recovery

Undo and Redo Operations –

Because all database modifications must be preceded by creation of log record, the system has available both the old value prior to modification of data item and new value that is to be written for data item.

This allows system to perform redo and undo operations as appropriate:

1. **Undo:** using a log record sets the data item specified in log record to old value.
2. **Redo:** using a log record sets the data item specified in log record to new value.

The database can be modified using two approaches –

1. **Deferred Modification Technique**
2. **Immediate Modification Technique**

Log Based Recovery

The database can be modified using two approaches –

1. **Deferred Modification Technique**

- The deferred modification technique occurs if the transaction does not modify the database until it has committed.
- In this method, all the logs are created and stored in the stable storage, and the database is updated when a transaction commits.

2. **Immediate Modification Technique**

- The Immediate modification technique occurs if database modification occurs while the transaction is still active.
- In this technique, the database is modified immediately after every operation. It follows an actual database modification.

Recovery using Log records

When the system is crashed, then the system consults the log to find which transactions need to be undone and which need to be redone.

1. If the log contains the record <Ti, Start> and <Ti, Commit> or <Ti, Commit>, then the Transaction Ti needs to be redone.

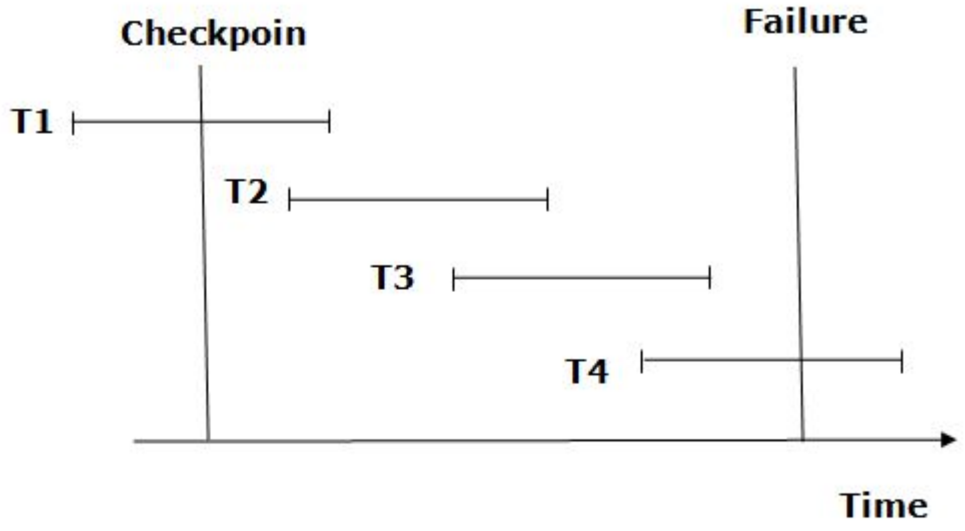
2. If log contains record<T_n, Start> but does not contain the record either <Ti, commit> or <Ti, abort>, then the Transaction Ti needs to be undone.

Recovery using Log records

Checkpoint

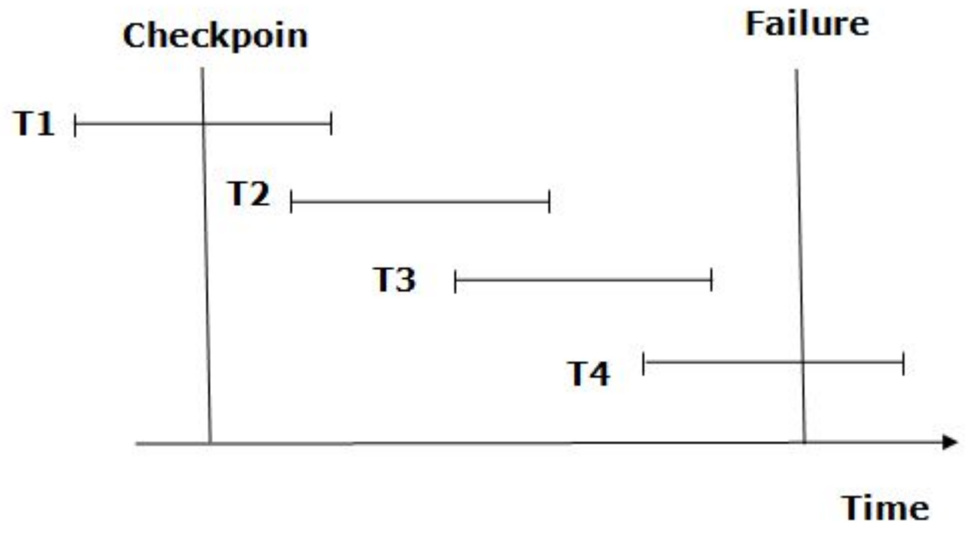
- The checkpoint is a type of mechanism where all the previous logs are removed from the system and permanently stored in the storage disk.
- The checkpoint is like a bookmark. While the execution of the transaction, such checkpoints are marked, and the transaction is executed then using the steps of the transaction, the log files will be created.
- When it reaches to the checkpoint, then the transaction will be updated into the database, and till that point, the entire log file will be removed from the file. Then the log file is updated with the new step of transaction till next checkpoint and so on.
- The checkpoint is used to declare a point before which the DBMS was in the consistent state, and all transactions were committed.

Recovery using Checkpoints



- The recovery system reads log files from the end to start. It reads log files from T4 to T1.
- Recovery system maintains two lists, a redo-list, and an undo-list.
- The transaction is put into redo state if the recovery system sees a log with <Tn, Start> and <Tn, Commit> or just <Tn, Commit>. In the redo-list and their previous list, all the transactions are removed and then redone before saving their logs.

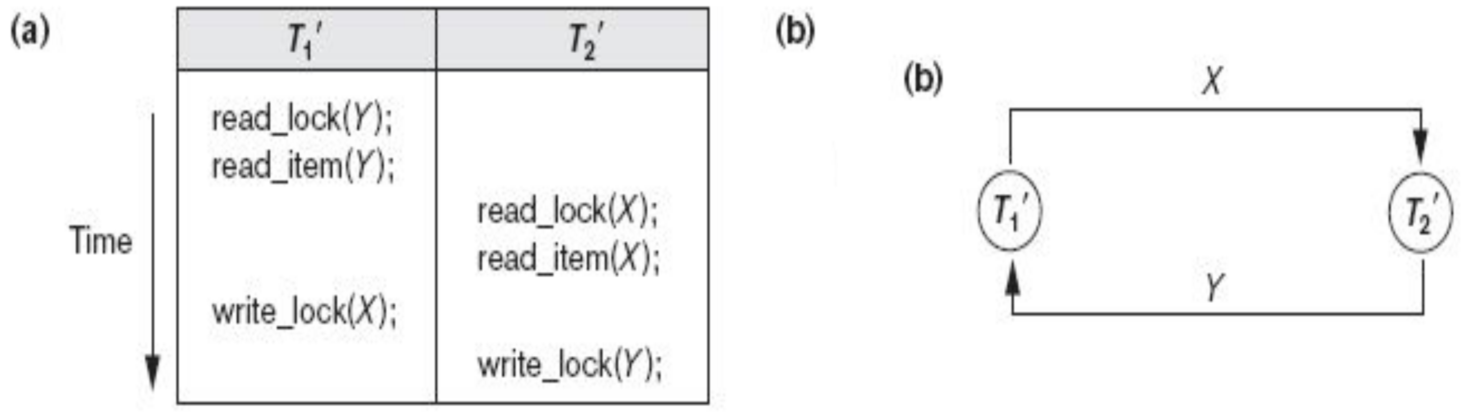
Recovery using Checkpoints



- **For example:** In the log file, transaction T2 and T3 will have <Tn, Start> and <Tn, Commit>. The T1 transaction will have only <Tn, commit> in the log file. That's why the transaction is committed after the checkpoint is crossed. Hence it puts T1, T2 and T3 transaction into redo list.
- The transaction is put into undo state if the recovery system sees a log with <Tn, Start> but no commit or abort log found. In the undo-list, all the transactions are undone, and their logs are removed.
- **For example:** Transaction T4 will have <Tn, Start>. So T4 will be put into undo list since this transaction is not yet complete and failed amid.

<u>DEADLOCK</u>

Deadlock occurs when each transaction T in a set of two or more transactions is waiting for some item X, but X is locked by another transaction T ’ in the set. Hence, each transaction in the set is on a waiting queue, waiting for one of the other transactions in the set to release the lock on an item.



T_1'	T_2'
read_lock(Y);	
read_item(Y);	
	read_lock(X);
	read_item(X);
write_lock(X);	
	write_lock(Y);

Figure 22.5
Illustrating the deadlock problem. (a) A partial schedule of T_1' and T_2' that is in a state of deadlock. (b) A wait-for graph for the partial schedule in (a).

DEADLOCK PREVENTION:

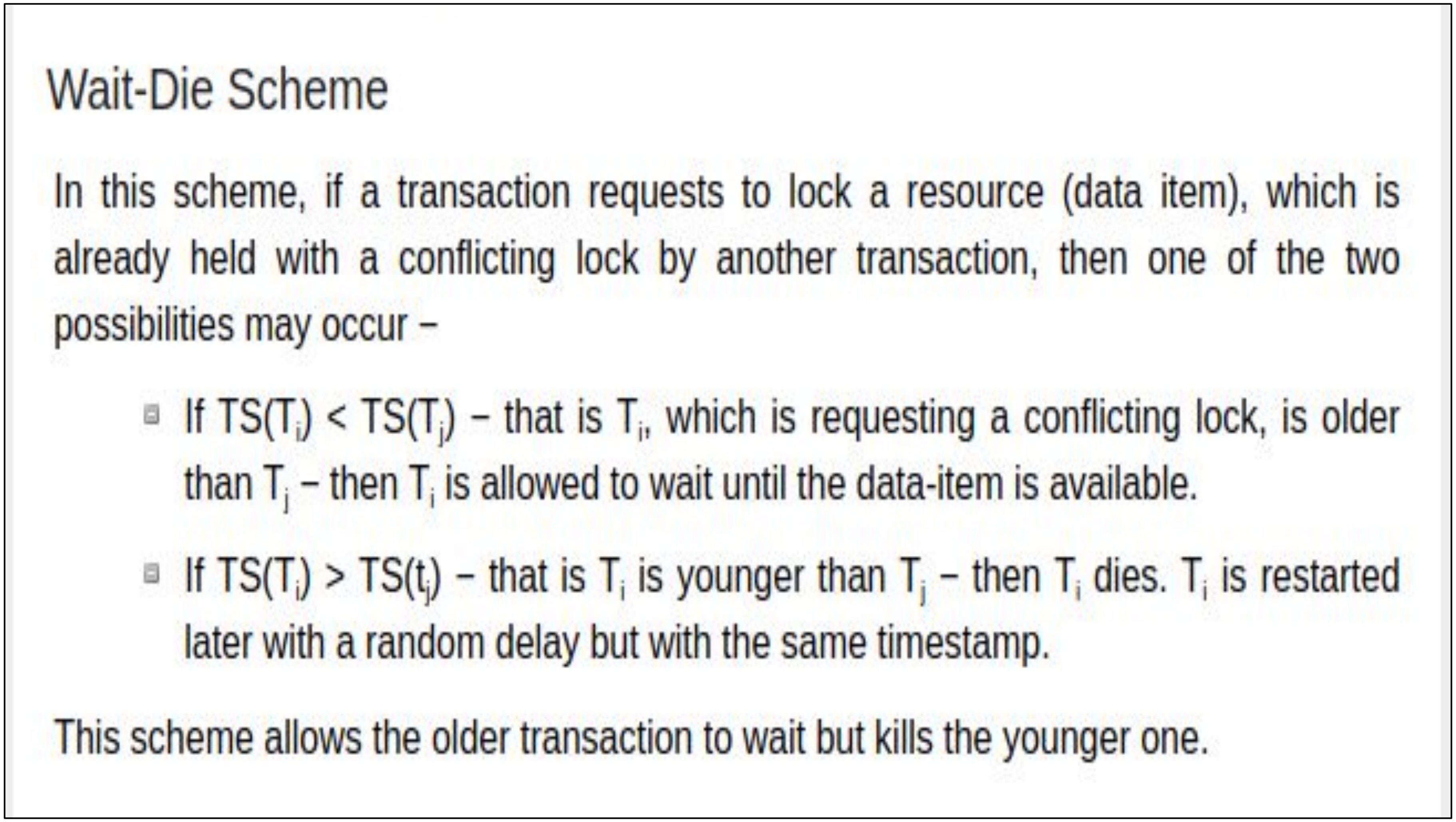
1. **Use Conservative Locking** : every transaction *lock all the items it needs in advance* (generally not a practical assumption) —if any of the items cannot be obtained, none of the items are locked. Rather, the transaction waits and then tries again to lock all the items it needs.

2. **Ordering all the items** in the database and making sure that a transaction that needs several items will lock them according to that order

3. **Use of transaction timestamp** TS(T), which is a unique identifier assigned to each transaction. The timestamps are typically ordered based on the order in which transactions are started; hence, if transaction T1 starts before transaction T2, then TS(T1) < TS(T2). Notice that the *older* transaction T1 has the *smaller* timestamp value.

Two Schemes that prevent deadlock are:

- Wait-Die
- Wound-Wait

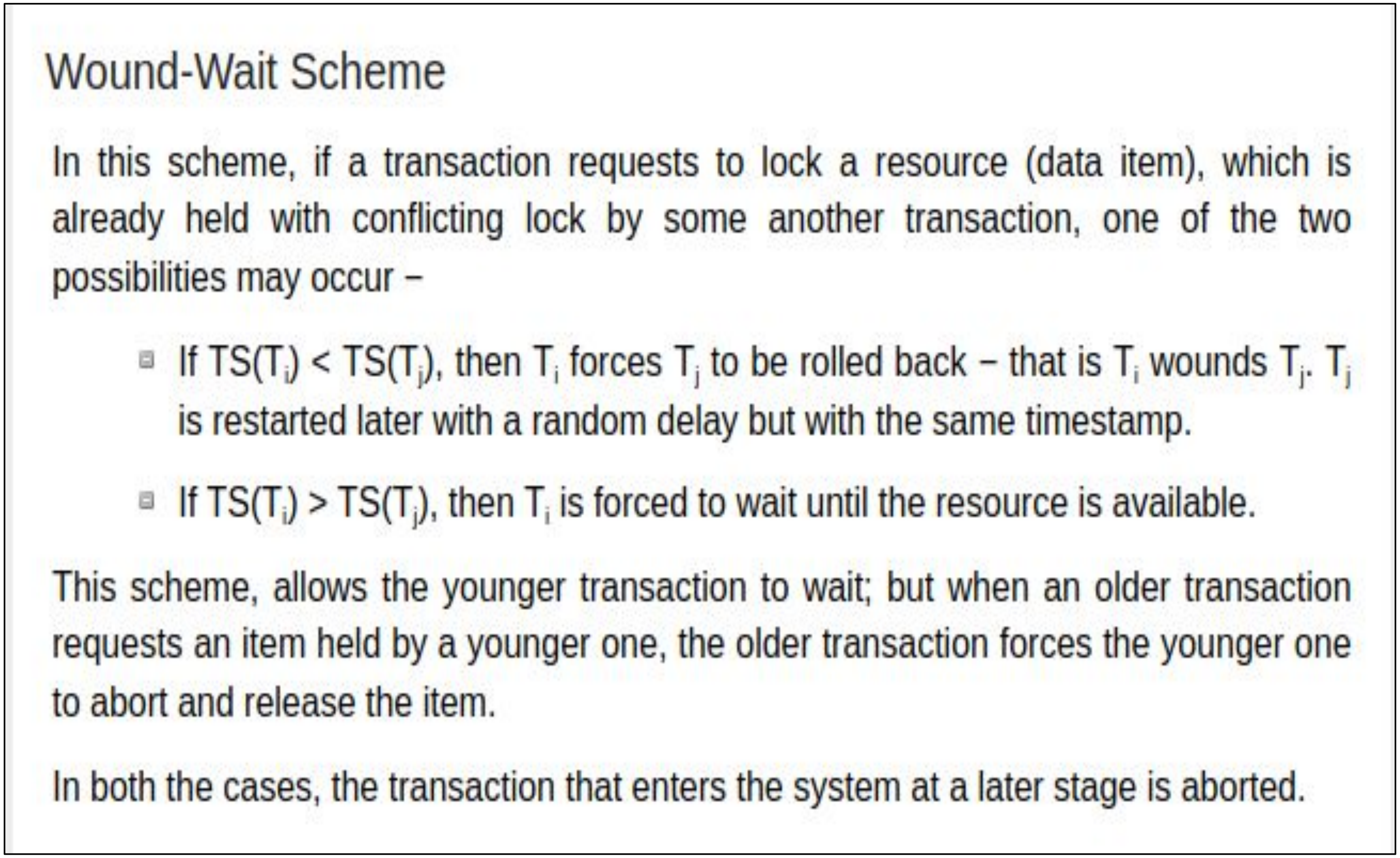


Wait-Die Scheme

In this scheme, if a transaction requests to lock a resource (data item), which is already held with a conflicting lock by another transaction, then one of the two possibilities may occur –

- If $TS(T_i) < TS(T_j)$ – that is T_i, which is requesting a conflicting lock, is older than T_j – then T_i is allowed to wait until the data-item is available.
- If $TS(T_i) > TS(t_j)$ – that is T_i is younger than T_j – then T_i dies. T_i is restarted later with a random delay but with the same timestamp.

This scheme allows the older transaction to wait but kills the younger one.



Wound-Wait Scheme

In this scheme, if a transaction requests to lock a resource (data item), which is already held with conflicting lock by some another transaction, one of the two possibilities may occur –

- If $TS(T_i) < TS(T_j)$, then T_i forces T_j to be rolled back – that is T_i wounds T_j. T_j is restarted later with a random delay but with the same timestamp.
- If $TS(T_i) > TS(T_j)$, then T_i is forced to wait until the resource is available.

This scheme, allows the younger transaction to wait; but when an older transaction requests an item held by a younger one, the older transaction forces the younger one to abort and release the item.

In both the cases, the transaction that enters the system at a later stage is aborted.

DEADLOCK PREVENTION (Contd.)

4. No waiting: In case of inability to obtain a lock, a transaction aborts and is resubmitted with a fixed delay. (Causes too many needless aborts).

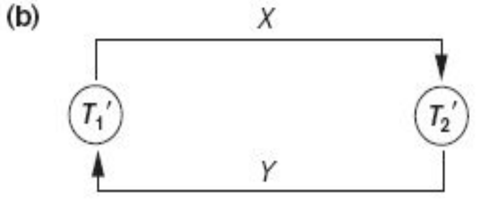
5. Cautious Waiting :

Suppose that transaction Ti tries to lock an item X but is not able to do so because X is locked by some other transaction Tj with a conflicting lock.

- **cautious waiting**: if Tj is not blocked (not waiting for some other locked item) then Ti is blocked and allowed to wait otherwise abort Ti

DEADLOCK DTECTION

1. **DRAW WAIT – FOR - GRAPH**



2. Use Of Timeouts:

- Practical because of its low overhead and simplicity.
- If a transactions waits for a period longer than a system-defined timeout period, the system assumes that the transaction may be deadlocked and aborts it—regardless of whether a deadlock actually exists or not.

- DISADVANTAGE: Some transactions that were not deadlocked may abort and may have to be resubmitted

Starvation

1. A transaction is **starved** if it cannot proceed for an indefinite period of time while other transactions in the system continue normally. This may occur if the waiting scheme for locked items is unfair, giving priority to some transactions over others.

Solution : have a fair waiting scheme(Eg. FCFS basis)

2. Starvation can also occur in the algorithms for dealing with deadlock. Occurs if the algorithms select the same transaction as victim repeatedly, thus causing it to abort and never finish execution.

Solution : wait-die and wound – wait avoid starvation

REMEDIES FOR PREVENTING STARVATION

1. first-come-first-serve queue
- a fair waiting scheme;
-transactions are enabled to lock an item in the order in which they originally requested to lock the item.

2. allow some transactions to have priority over others but increase the priority of a transaction the longer it waits, until it eventually gets the highest priority and proceeds.

3. The victim selection algorithm can use higher priorities for transactions that have been aborted multiple times so that they are not selected as victims repeatedly.

The wait-die and wound-wait schemes avoid starvation.

<u>Thank You</u>