# Database Management Systems

Fall 2001

## CMPUT 391: Transactions & Concurrency Control

Dr. Osmar R. Zaïane

University of Alberta

Chapters 18 and 19 of Textbook

---

# Course Content

- Introduction
- Database Design Theory
- Query Processing and Optimisation
- **Concurrency Control**
- Data Base Recovery and Security
- Object-Oriented Databases
- Inverted Index for IR
- XML
- Data Warehousing
- Data Mining
- Parallel and Distributed Databases
- Other Advanced Database Topics

---

# Objectives of Lecture 4
## Transactions and Concurrency Control

- Introduce some important notions related to DBMSs such as transactions, scheduling, locking mechanisms, committing and aborting transactions, etc.
- Understand the issues related to concurrent execution of transactions on a database.
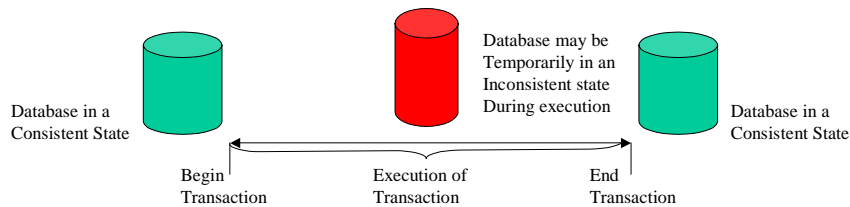- Present some typical anomalies with interleaved executions.

---

# Transactions and Concurrency Control

- Transactions in a Database
- Transaction Processing
- Schedules and Serializability
- Concurrency Control Techniques
- Locking Mechanisms and Timestamps

# Transaction

- A transaction is the DBMS's abstract view of a user program: a sequence of reads and writes

- A transaction is a sequence 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
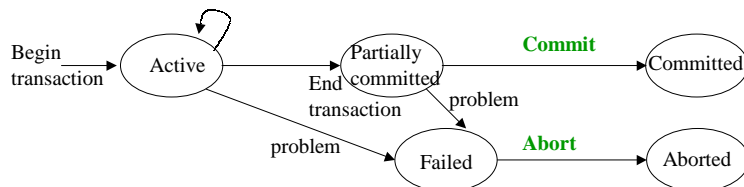
End Transaction

# Transaction Operations

- A user's program may carry out many operations on the data retrieved from DB but DBMS is only concerned about Read/Write.

- A database transaction is the execution of a program that include database access operations:
  - Begin-transaction
  - Read
  - Write
  - End-transaction
  - Commit-transaction
  - Abort-transaction
  - Undo
  - Redo

- Concurrent execution of user programs is essential for good DBMS performance.

# State of Transactions

- **Active**: the transaction is executing.
- **Partially Committed**: the transaction ends after execution of final statement.
- **Committed**: after successful completion checks.
- **Failed**: when the normal execution can no longer proceed.
- **Aborted**: after the transaction has been rolled back.

Begin transaction → Active

Active → Partially committed (End transaction)

Partially committed → **Commit** → Committed

Partially committed → problem → Failed

Active → problem → Failed

Failed → **Abort** → Aborted

# Concurrency in a DBMS

- Users submit transactions, and can think of each transaction as executing by itself.
  - Concurrency is achieved by the DBMS, which interleaves actions (reads/writes of DB objects) of various transactions.
  - Each transaction must leave the database in a consistent state if the DB is consistent when the transaction begins.
    - DBMS will enforce some ICs, depending on the ICs declared in CREATE TABLE statements.
    - Beyond this, the DBMS does not really understand the semantics of the data. (e.g., it does not understand how the interest on a bank account is computed).

- *Issues:* Effect of *interleaving* transactions, and *crashes.*

# Transactions and Concurrency Control

- Transactions in a Database
- Transaction Processing
- Schedules and Serializability
- Concurrency Control Techniques
- Locking Mechanisms and Timestamps

# Transaction Properties

The acronym ACID is often used to refer to the four properties of DB transactions.

- **A**tomicity (all or nothing)
  - A transaction is *atomic*: transaction always executing all its actions in one step, or not executing any actions at all.
- **C**onsistency (no violation of integrity constraints)
  - A transaction must preserve the consistency of a database after execution. (responsibility of the user)
- **I**solation (concurrent changes invisible➔serializable)
  - Transaction is protected from the effects of concurrently scheduling other transactions.
- **D**urability (committed updates persist)
  - The effect of a committed transaction should persist even after a crash.

# Atomicity

- Either all or none of the transaction's operations are performed.
- Atomicity requires that if a transaction is interrupted by a failure, its partial results must be **undone**.
- The activity of preserving the transaction's atomicity in presence of transaction' aborts due to input errors, system overloads, or deadlocks is called **transaction recovery**.
- The activity of ensuring atomicity in the presence of system crashes is called **crash recovery**.
  (will be discussed in the next lecture)

# Consistency

- A transaction which executes *alone* against a consistent database leaves it in a *consistent* state.
- Transactions do not violate database integrity constraints.
- Transactions are *correct* programs

# Isolation

- If several transactions are executed concurrently, the results must be the same as if they were executed serially in some order (serializability).

- An incomplete transaction cannot reveal its results to other transactions before its commitment.

- Necessary to avoid cascading aborts.

# Durability

- Once a transaction commits, the system must guarantee that the result of its operations will never be lost, in spite of susequent failures.

- Database recovery (will be discussed in the next lecture)

# Example

- Consider two transactions:

```
T1:    BEGIN   A=A+100,  B=B-100   END
T2:    BEGIN   A=1.06*A,  B=1.06*B  END
```

- ❖ Intuitively, the first transaction is transferring $100 from B's account to A's account. The second is crediting both accounts with a 6% interest payment.
- ❖ There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together.
- ❖ **However, the net effect *must* be equivalent to these two transactions running serially in some order.**

# Example (Contd.)

- Consider a possible interleaving (*schedule*):

```
T1:      A=A+100,                      B=B-100
T2:               A=1.06*A,                      B=1.06*B
```

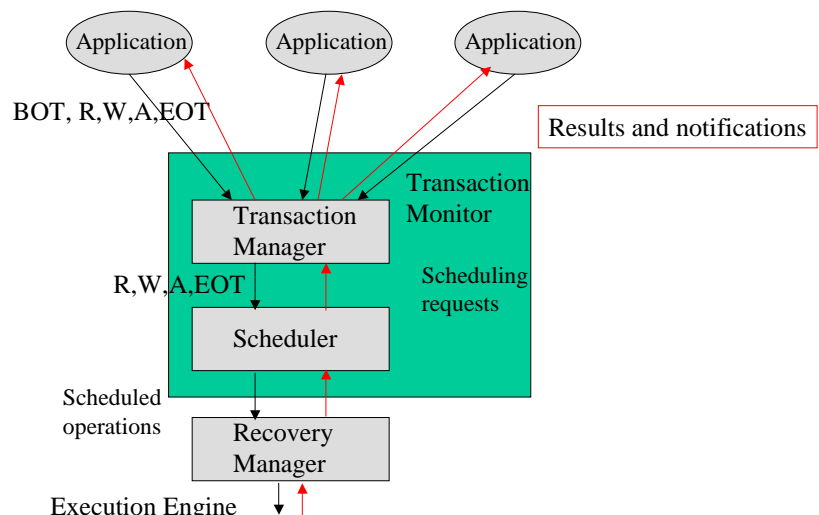- ❖ This is OK. But what about:

```
T1:      A=A+100,                               B=B-100
T2:               A=1.06*A, B=1.06*B
```

- ❖ The DBMS's view of the second schedule:

```
T1:     R(A), W(A),                              R(B), W(B)
T2:               R(A), W(A), R(B), W(B)
```

## Transaction Execution

## Transactions and Concurrency Control

- Transactions in a Database
- Transaction Processing
- Schedules and Serializability
- Concurrency Control Techniques
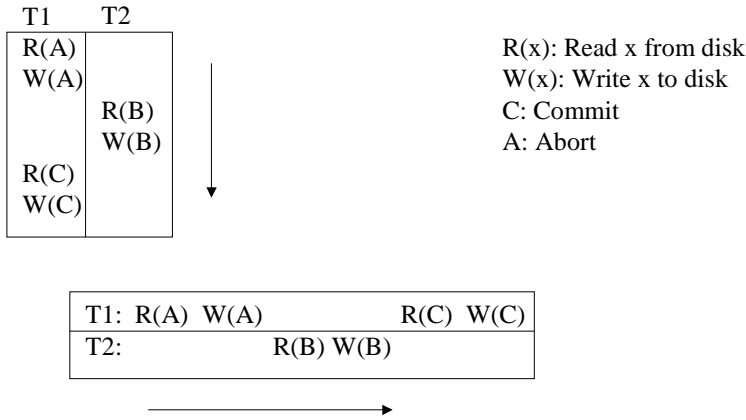- Locking Mechanisms and Timestamps

## Scheduling Transactions

- *A Schedule* is a sequential order of the instructions (R / W / A / C) of *n* transactions such that the ordering of the instructions of each transaction is preserved. (execution sequence preserving the operation order of individual transaction)
- *Serial schedule:* A schedule that does not interleave the actions of different transactions. (transactions executed consecutively)
- *Non-serial schedule*: A schedule where the operations from a set of concurrent transactions are interleaved.

S1 
| T1: | A=A+100, | | B=B-100 | |
| T2: | | A=1.06*A, | | B=1.06*B |

S2 
| T1: | A=A+100, B=B-100 | |
| T2: | | A=1.06*A,B=1.06*B |

## Scheduling Transactions (continue)

- *Equivalent schedules*: For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.
- *Serializable schedule*: A non-serial schedule that is equivalent to some serial execution of the transactions.

  (Note: If each transaction preserves consistency, every serializable schedule preserves consistency. )
- Two schedules are conflict equivalent if:
  - Involve the same actions of the same transactions
  - Every pair of conflicting actions is ordered the same way
- Schedule S is conflict serializable if S is conflict equivalent to some serial schedule

## Schedule Conventions

| T1 | T2 |
|----|----|
| R(A) | |
| W(A) | |
| | R(B) |
| | W(B) |
| R(C) | |
| W(C) | |

R(x): Read x from disk
W(x): Write x to disk
C: Commit
A: Abort

| T1: R(A)  W(A) | R(C)  W(C) |
|---|---|
| T2: | R(B) W(B) |

## Conflicts of Operations

• If two transactions only read a data object, they do not conflict and the order is not important
• If two transactions either read or write completely separate data objects, they do not conflict and the order is not important.
• If one transaction writes a data object and another either reads or writes the same data object, the order of execution is important.

| | Read(x) | Write(x) |
|----------|---------|----------|
| Read(x) | No | Yes |
| Write(x) | Yes | Yes |

WR conflict: T2 reads a data objects previously written by T1
RW conflict: T2 writes a data object previously read by T1
WW conflict: T2 writes a data object previously written by T1

## Anomalies with Interleaved Execution

• Reading Uncommitted Data (WR Conflicts, "dirty reads": read an object modified by uncommited transaction.):

Aka:
Uncommitted Dependency
Dirty read problem

• T1 transfers $100 from A to B
• T2 adds 6% to A and B
• Avoid cascading aborts

| T1: R(A), W(A), | R(B), W(B), Abort |
|---|---|
| T2: | R(A), W(A),R(B),W(B), C |

## Anomalies (Continued)

• Unrepeatable Reads (RW Conflicts):

T1 tries to read a data object again after T2 modified it. The data object may have a different value.

| T1: R(A), | R(A), W(A), C |
|---|---|
| T2: R(A), W(A), C | |

Also,
T1 reads A and add 1. T2 Reads A and subtracts 1. If A initially 5, result should be 5
However:  T1: R(A) A+1                 W(A)
             T2:        R(A) A-1  W(A)

## Anomalies (Continued)

- Overwriting Uncommitted Data (WW Conflicts):
  - T1 sets salaries to $1000 and T2 sets salaries to $2000
  - Constraint: Salaries must be kept equal.

```
T1:     W(A),                    W(B), C
T2:             W(A), W(B), C
```

## The Inconsistent Analysis Problem

- Occurs when a transaction reads several values from a database while a second transaction updates some of them.

| T1 | T2 | A | B | C | sum |
|----|----|----|----|----|----|
| sum=0 | | $100 | $50 | $25 | 0 |
| R(A) | R(A) | $100 | $50 | $25 | 0 |
| sum=sum+A | A=A-10 | $100 | $50 | $25 | 100 |
| R(B) | W(A) | $90 | $50 | $25 | 100 |
| sum=sum+B | R(C) | $90 | $50 | $25 | 150 |
| | C=C+10 | $90 | $50 | $25 | 150 |
| | W(C) | $90 | $50 | $35 | 150 |
| R(C) | | $90 | $50 | $35 | 150 |
| sum=sum+C | | $90 | $50 | $35 | 185 |

Should be 175

## Serializability

- The objective of *serializability* is to find non-serial schedules that allow transactions to execute concurrently without interfering with one another, and thereby produce a database state that could be produced by a serial execution.
- It is important to guarantee serializability of concurrent transactions in order to prevent inconsistency from transactions interfering with one another.
- In serializability, the ordering of read and write operations is important (see conflict of operations).
- See the following schedules how the order of R/W operations can be changed depending upon the data objects they relate to.

## Schedules Example

| T1 | T2 | S1 |
|----|----|----|
| R(A) | | |
| W(A) | | |
| | R(A) | |
| | W(A) | |
| R(B) | | |
| W(B) | | |
| Commit | | |
| | R(B) | |
| | W(B) | |
| | Commit | |

| T1 | T2 | S2 |
|----|----|----|
| R(A) | | |
| W(A) | | |
| | R(A) | |
| R(B) | | |
| | W(A) | |
| W(B) | | |
| Commit | | |
| | R(B) | |
| | W(B) | |
| | Commit | |

| T1 | T2 | S3 |
|----|----|----|
| R(A) | | |
| W(A) | | |
| R(B) | | |
| W(B) | | |
| Commit | | |
| | R(A) | |
| | W(A) | |
| | R(B) | |
| | W(B) | |
| | Commit | |

In S2: change the order of W(A) in T2 with W(B) in T1
In S2: change the order of R(A) in T2 with R(B) in T1    ➔ S3
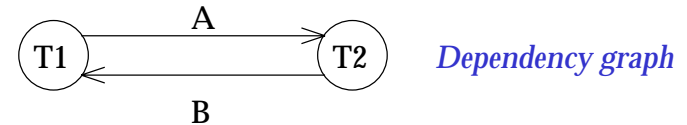In S2: change the order of ((A) in T2 with W(B) in T1

# Dependency Graph

- *Dependency graph*:
  - One node per transaction;
  - edge from *Ti* to *Tj* if *Tj* reads/writes an object last written by *Ti*.

- <u>Theorem</u>: Schedule is conflict serializable if and only if its dependency graph is acyclic

# Example

- A schedule that is not conflict serializable:

| T1: | R(A), W(A), | | R(B), W(B) |
|---|---|---|---|
| T2: | | R(A), W(A), R(B), W(B) | |



*Dependency graph*

- The cycle in the graph reveals the problem. The output of T1 depends on T2, and vice-versa.

# Algorithm for Testing Serializability of a Schedule S

1. For each transaction $T_i$ in S
   create a node labeled $T_i$ in the precedence graph.
2. For each case in S where $T_j$ executes a Read(x) after a Write(x) executed by $T_i$
   create an edge $(T_i, T_j)$ in the precedence graph
3. For each case in S where $T_j$ executes a Write(x) after a Read(x) executed by $T_i$
   create an edge $(T_i, T_j)$ in the precedence graph
4. For each case in S where $T_j$ executes a Write(x) after a Write(x) executed by $T_i$
   create an edge $(T_i, T_j)$ in the precedence graph
5. S is serializable iff the precedence graph has no cycles

# Transactions and Concurrency Control

- Transactions in a Database
- Transaction Processing
- Schedules and Serializability
- Concurrency Control Techniques
- Locking Mechanisms and Timestamps

# Serializability in Practice

- In practice, a DBMS does not test for serializability of a given schedule. This would be impractical since the interleaving of operations from concurrent transactions could be dictated by the OS and thus could be difficult to impose.

- The approach take by the DBMS is to use specific protocols that are known to produce serializable schedules.

- These protocols could reduce the concurrency but eliminate conflicting cases.

# Lock-Based Concurrency Control

- *Strict Two-phase Locking (Strict 2PL) Protocol*:
  - Each transaction must obtain a S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing.
  - All locks held by a transaction are released when the transaction completes
  - If an transaction holds an X lock on an object, no other transaction can get a lock (S or X) on that object.
- Strict 2PL allows only serializable schedules.

# Aborting a Transaction

- If a transaction $T_i$ is aborted, all its actions have to be <u>undone</u>. Not only that, if $T_j$ reads an object last written by $T_i$, $T_j$ must be aborted as well!

- Most systems avoid such *cascading aborts* by releasing a transaction's locks only at commit time.
  - If $T_i$ writes an object, $T_j$ can read this only after $T_i$ commits.

- In order to *undo* the actions of an aborted transaction, the DBMS maintains a *log* in which every write is recorded. This mechanism is also used to recover from system crashes: all active transactions at the time of the crash are aborted when the system comes back up.

# The Log

- The following actions are recorded in the log:
  - *$T_i$ writes an object*: the old value and the new value.
    - Log record must go to disk *before* the changed page!
  - *$T_i$ commits/aborts*: a log record indicating this action.
- Log records are chained together by transaction id, so it's easy to undo a specific transaction.
- Log is often *duplexed* and *archived* on stable storage.
- All log related activities (and in fact, all CC related activities such as lock/unlock, dealing with deadlocks etc.) are handled transparently by the DBMS.

## Recovering From a Crash

- There are 3 phases in the *Aries* recovery algorithm:
  - *Analysis*: Scan the log forward (from the most recent *checkpoint*) to identify all transactions that were active, and all dirty pages in the buffer pool at the time of the crash.
  - *Redo*: Redoes all updates to dirty pages in the buffer pool, as needed, to ensure that all logged updates are in fact carried out and written to disk.
  - *Undo*: The writes of all transactions that were active at the crash are undone (by restoring the *before value* of the update, which is in the log record for the update), working backwards in the log. (Some care must be taken to handle the case of a crash occurring during the recovery process!)

  More details in next Lecture

## Transactions and Concurrency Control

- Transactions in a Database
- Transaction Processing
- Schedules and Serializability
- Concurrency Control Techniques
- Locking Mechanisms and Timestamps

## Concurrency Control Algorithms

- **Pessimistic (or Conservative) Approach**

  Cause transactions to be delayed in case they conflict with other transactions at some time in the future
  - **Two-Phase Locking (2PL)**
  - **Timestamp Ordering (TO)**

- **Optimistic Approach**

  Allow transactions to proceed unsynchronized and only check conflicts at the end

  (based on the premise that conflicts are rare)

## Pessimistic vs. Optimistic

- Pessimistic Execution

  | Validate | Read | *Compute* | Write |

- Optimistic Execusion

  | Read | *Compute* | Validate | Write |

- Optimistic CC Validation Test

  Validation succeeds for all transaction $T_k$ and $T_i$ where $ts(T_k) < ts(T_i)$ and $T_k$ start write before $T_i$ start read.

  $T_k$ | R | C | V | W |
        $T_i$ | R | C | V | W |

  Validation succeeds for all transaction $T_k$ and $T_i$ where $ts(T_k) < ts(T_i)$ and $T_k$ and $T_i$ don't access common data. $W(T_k) \cap R(T_i) = \varnothing$ and $W(T_k) \cap W(T_i) = \varnothing$

  $T_k$ | R | C | V | W |
        $T_i$ | R | C | V | W |

# Definitions

- **Locking**:  A procedure used to control concurrent access to data. When one transaction is accessing the database, a lock may deny access to other transactions to prevent incorrect results.
- **Shared Lock** (or read lock): If a transaction has a shared lock on a data object, it can read the object but not update it.
- **Exclusive Lock** (or write lock): if a transaction has an exclusive lock on a data object, it can both read and update the object.

# Locking-Based Algorithms

- Transactions indicate their intensions by requesting locks from the scheduler (lock manager).
- Every transaction that needs to access a data object for reading or writing must first lock the object.
- A transaction holds a lock until it explicitly releases it.
- Locks are either shared or exclusive.
- Shared and exclusive locks conflict

|           | Shared | Exclusive |
|-----------|--------|-----------|
| Shared    | Yes    | No        |
| Exclusive | No     | No        |

Compatibility

- Locks allow concurrent processing of transactions.

# Two-Phase Locking

- A transaction follows the 2PL protocol if all locking operations precede the first unlock operation in the transaction.
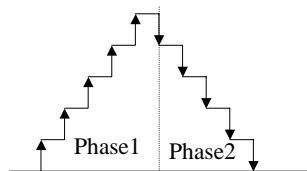
  - Phase 1 is the "growing phase" during which all the locks are requested
  - Phase 2 is the "shrinking phase" during which all locks are released

  1. A transaction locks an object before using it
  2. When an object is already locked by another transaction, the requesting transaction must wait until the lock is released
  3. When a transaction releases a lock, it may not request another lock.

Phase1   Phase2

# Strict Two-Phase Locking

- Transaction holds locks until the end of transaction (just before committing)
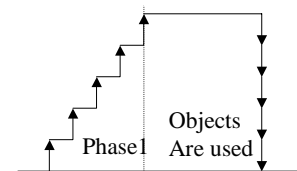
a.k.a.
Conservative 2PL

Phase1   Objects Are used

# Lock Management

- Lock and unlock requests are handled by the lock manager
- Lock table entry:
  - Number of transactions currently holding a lock
  - Type of lock held (shared or exclusive)
  - Pointer to queue of lock requests
- Locking and unlocking have to be atomic operations
- Lock upgrade: (for some DBMSs) transaction that holds a shared lock can be upgraded to hold an exclusive lock (also downgrade)
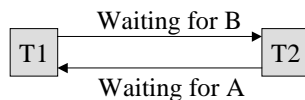
# Deadlocks

- Deadlock: Cycle of transactions waiting for locks to be released by each other.
- A transaction is deadlocked if it is blocked and will remain blocked until intervention.
- Locking-based Concurrency Control algorithms may cause deadlocks.
- Two ways of dealing with deadlocks:
  - Deadlock prevention (guaranteeing no deadlocks or detecting deadlocks in advance before they occur)
  - Deadlock detection (allowing deadlocks to form and breaking them when they occur)

# Deadlock Example

```
T1                      T2
begin-transaction
Write-lock(A)           begin-transaction
Read(A)                 Write-lock(B)
A=A-100                 Read(B)
Write(A)                B=B*1.06
Write-lock(B)           Write(B)
Wait                    write-lock(A)
Wait                    Wait
…                       Wait
                        …
```

Waiting for B

T1 → T2

Waiting for A

# Deadlock Prevention

- Assign priorities based on timestamps (i.e. The oldest transaction has higher priority).
- Assume $T_i$ wants a lock that $T_j$ holds. Two policies are possible:
  - Wait-Die: If $T_i$ has higher priority, $T_i$ allowed to wait for $T_j$; otherwise ($T_i$ younger) $T_i$ aborts
  - Wound-wait: If $T_i$ has higher priority, $T_j$ aborts; otherwise ($T_i$ younger) $T_i$ waits
- If a transaction re-starts, make sure it has its original timestamp

# Deadlock and Timeouts

- A simple approach to deadlock prevention (and pseudo detection) is based on lock timeouts

- After requesting a lock on a locked data object, a transaction waits, but if the lock is not granted within a period (timeout), a deadlock is assumed and the waiting transaction is aborted and re-started.

- Very simple practical solution adopted by many DBMSs.

---

# Deadlock Detection

- Create a waits-for graph:
  - Nodes are transactions
  - There is an edge from $T_i$ to $T_j$ if $T_i$ is waiting for $T_j$ to release a lock

- Deadlock exists if there is a cycle in the graph.

- Periodically check for cycles in the waits-for graph.

---

# Deadlock Detection (Continued)

Example:

```
T1:  S(A), R(A),              S(B)
T2:            X(B),W(B)                X(C)
T3:                      S(C), R(C)          X(A)
T4:                                    X(B)
```

---
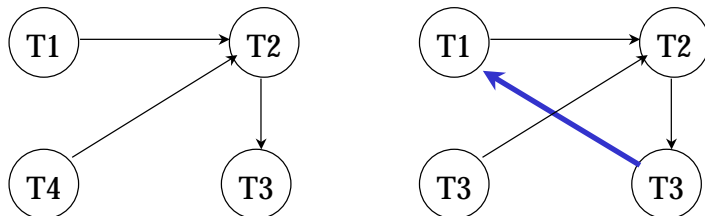
# Recovery from Deadlock

- How to choose a a deadlock victim to abort?
  - How long the transaction has been running?
  - How many data objects have been updated?
  - How many data objects the transaction is still to update?

- Do we need to rollback the whole aborted transaction?

- Avoid starvation (when the same transaction is always the victim)

# Timestamping

- Each transaction is assigned a globally unique timestamp (starting time)
- A concurrency control protocol that orders transactions in such a way that older transactions get priority in the event of conflict.
- Each data object is assigned
  - a write timestamp wts (largest timestamp on any write on x) and
  - a read timestamp rts (largest timestamp on any read on x)
- Conflict operations are resolved by timestamp ordering.

# Timestamp Ordering

- For $R_i(x)$
  - if $ts(T_i) < wts(x)$ then reject $R_i(x)$
  - else accept $R_i(x)$; $rts(x) \leftarrow ts(T_i)$

- For $W_i(x)$
  - if $ts(T_i) < rts(x)$ or $ts(T_i) < wts(x)$ then reject $W_i(x)$
  - else accept $W_i(x)$; $wts(x) \leftarrow ts(T_i)$

# Summary

- Concurrency control and recovery are among the most important functions provided by a DBMS.
- Users need not worry about concurrency.
  - System automatically inserts lock/unlock requests and schedules actions of different transactios in such a way as to ensure that the resulting execution is equivalent to executing the transactions one after the other in some order.
- Write-ahead logging (WAL) is used to undo the actions of aborted transactions and to restore the system to a consistent state after a crash.
  - *Consistent state*:  Only the effects of commited transactions seen.

# Summary (Contd.)

- There are several lock-based concurrency control schemes (Strict 2PL, 2PL). Conflicts between transactions can be detected in the dependency graph
- The lock manager keeps track of the locks issued. Deadlocks can either be prevented or detected.
- Timestamp CC is another alternative to 2PL; allows some serializable schedules that 2PL does not (although converse is also true).
- Ensuring recoverability with Timestamp CC requires ability to block transactions, which is similar to locking.