The University of Manchester

H10

# Transactions, Concurrency and Recovery

Fundamentals of Databases

Alvaro A A Fernandes, SCS, UoM

[COMP23111 Handout 10 of 12 ]

## Acknowledgements

- These slides are adaptations (mostly minor, but some major) of material authored and made available to instructors by **Thomas Connolly and Carolyn Begg** to accompany their textbook **Database Systems: A Practical Approach to Design, Implementation, and Management, 5th Edition. Addison-Wesley, 2010, 978-0-321-52306-8**
- Copyright remains with them and the publishers, whom I thank.
- Some slides had input from Sandra Sampaio, whom I thank.
- All errors are my responsibility.
- Note that, for copyright reasons, the mandatory readings regarding the topics treated here come from **Michael Kifer, Arthur Bernstein and Philip M. Lewis, Database Systems: An Application-Oriented Approach, 2nd ed., Pearson Addison-Wesley, 2005.**

## In Previous Handouts

- We learned about the relational algebra and SQL, both its DDL and DML capabilities and its querying constructs.
- We also learned how to design databases for implementation in a relational DBMS.
- We explored some advanced features of SQL that can be used to capture more application semantics in the DBMS environment.
- In particular, we explored features of SQL that make it Turing-complete and lead to the notion of stored procedures and stored functions in DBMSs.
- We also explored triggers, a kind of construct the enables a DBMS to respond to events such a modification of the database instance.

## In This Handout

- We'll look into how DBMSs provide sophisticated services such as concurrent access, transaction management, and recovery from failures.
- We'll look into the lifecycle of a transaction, what properties transactions have and how these lead to strong requirements for concurrency control.
- We'll look into some of the problems that concurrency may give rise to and how to address them using such notions as serializability and recoverability of schedules.
- We'll look into deadlock detection and prevention in DBMSs as well as briefly discuss what recovery services they offer.

# Transactions, Concurrency, Recovery

- A DBMS provides services that ensure that users can rely on the database being a correct reflection of the state of the world it is designed to represent.
- This can be phrased in terms of the database being always a consistent representation of the (appropriate part of the) world.
- This reliability must be ensured in the presence of failures of either software or hardware even as multiple users are simultaneously making use of the DBMS to access the database.
- Three important DBMS services in this respect are:
  - transaction management
  - concurrency control
  - recovery management
- Although we will explore them one-by-one, they are mutually dependent.

## Transaction Support

- A transaction is an action, or series of actions, carried out by a user or application, which reads or updates the contents of a database.
- A transaction is the logical unit of work on the database.
- An application program can then be seen as a series of transactions with non-database processing in between.
- A transaction transforms (or transitions) a database from one consistent state to another, although consistency may be temporarily violated while the transaction is being executed.

## Example Transaction

```
read(staffNo = x, salary)
salary = salary * 1.1
write(staffNo = x, new_salary)
```

(a)

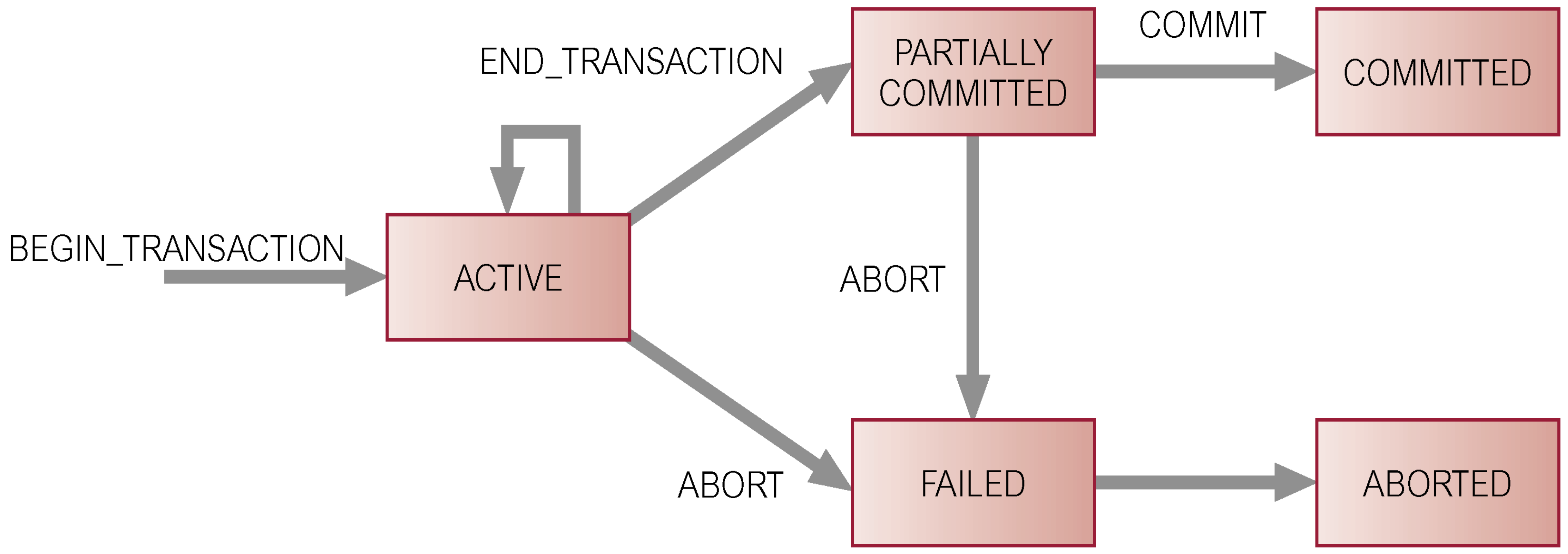
```
delete(staffNo = x)
for all PropertyForRent records, pno
begin
    read(propertyNo = pno, staffNo)
    if (staffNo = x) then
    begin
        staffNo = newStaffNo
        write(propertyNo = pno, staffNo)
    end
end
```

(b)

## Transaction Support

- A transaction can have one of two outcomes:
  - Success: the transaction commits and the database has thereby reached a new consistent state.
  - Failure: the transaction aborts, and database is thereby be restored to the consistent state before the transaction started.
- A failed transaction is rolled back or undone.
- A committed transaction cannot be aborted.
- An aborted transaction that is rolled back can be restarted later.

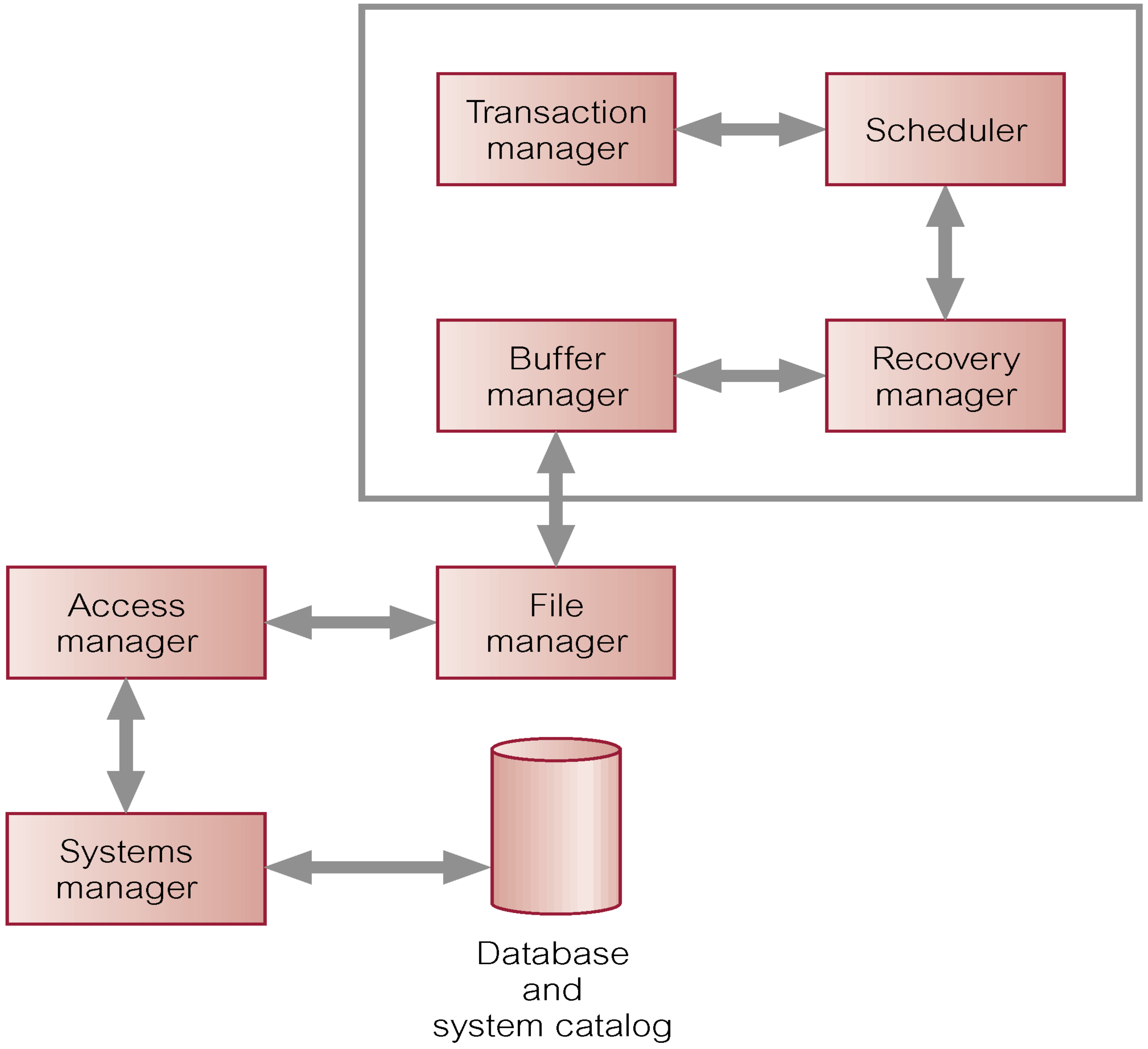
## State Transition Diagram for Transaction



# Properties of Transactions

- Transaction have four basic (called ACID) properties:
  - Atomicity: A transaction must either complete all its work or else it must be as if no partial work was ever done, i.e., an ‘all or nothing’ contract.
  - Consistency: A transaction must transform database from one consistent state to another.
  - Isolation: Partial effects of incomplete transactions should not be visible to other transactions.
  - Durability: The effects of a committed transaction are permanent and must not be lost because of later failure.

## DBMS Transaction Subsystem



## Concurrency Control

- Concurrency control is the process of managing simultaneous operations on the database without allowing them to interfere with one another.
- It prevents interference when two or more users are accessing database simultaneously and at least one is updating data.
- Although two transactions may be correct in themselves, i.e., when they execute without interference, the interleaving of their operations may produce an incorrect result.

## Need for Concurrency Control

- Three examples of potential problems caused by concurrency are:
  - The lost update problem
  - The uncommitted dependency problem
  - The inconsistent analysis problem

## Lost Update Problem

- A lost update problem occurs when a successfully completed update is overridden by another user.
- For example :
  - Transaction T1 withdraws £10 from an account with balance balx, which is initially £100.
  - Transaction T2 deposits £100 into same account.
  - Serially, the final balance would be £190.
  - However, if T1 and T2 are allowed to interleave freely the result may not be what we would have expected.

## Lost Update Problem

- In this example, the update by T2 is lost.
- The loss of the T2 update is avoided if T1 is prevented from reading balx until after T2 updates it.

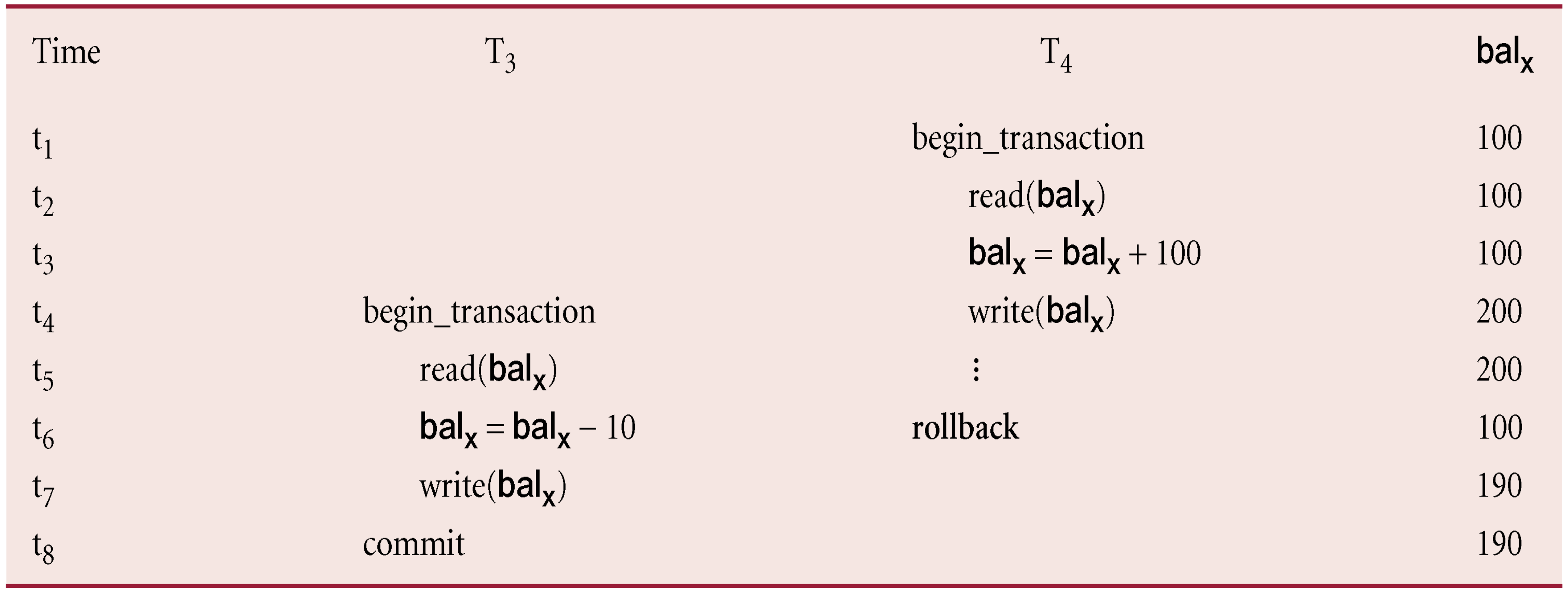
| Time | $T_1$ | $T_2$ | $bal_x$ |
|---|---|---|---|
| $t_1$ | | begin_transaction | 100 |
| $t_2$ | begin_transaction | read($bal_x$) | 100 |
| $t_3$ | read($bal_x$) | $bal_x = bal_x + 100$ | 100 |
| $t_4$ | $bal_x = bal_x - 10$ | write($bal_x$) | 200 |
| $t_5$ | write($bal_x$) | commit | 90 |
| $t_6$ | commit | | 90 |

## Uncommitted Dependency Problem

- An uncommitted dependency (or dirty read) problem occurs when one transaction can see the intermediate results of another transaction before it has committed.
- For example:
  - T4 updates balx to £200 but then aborts, so balx should be back at original value of £100.
  - However, T3 has read that new value of balx (£200) and uses that value as the basis of a £10 reduction, giving a new balance of £190, instead of £90.

## Uncommitted Dependency Problem

- The problem is avoided by preventing T3 from reading balx until after T4 commits or aborts.



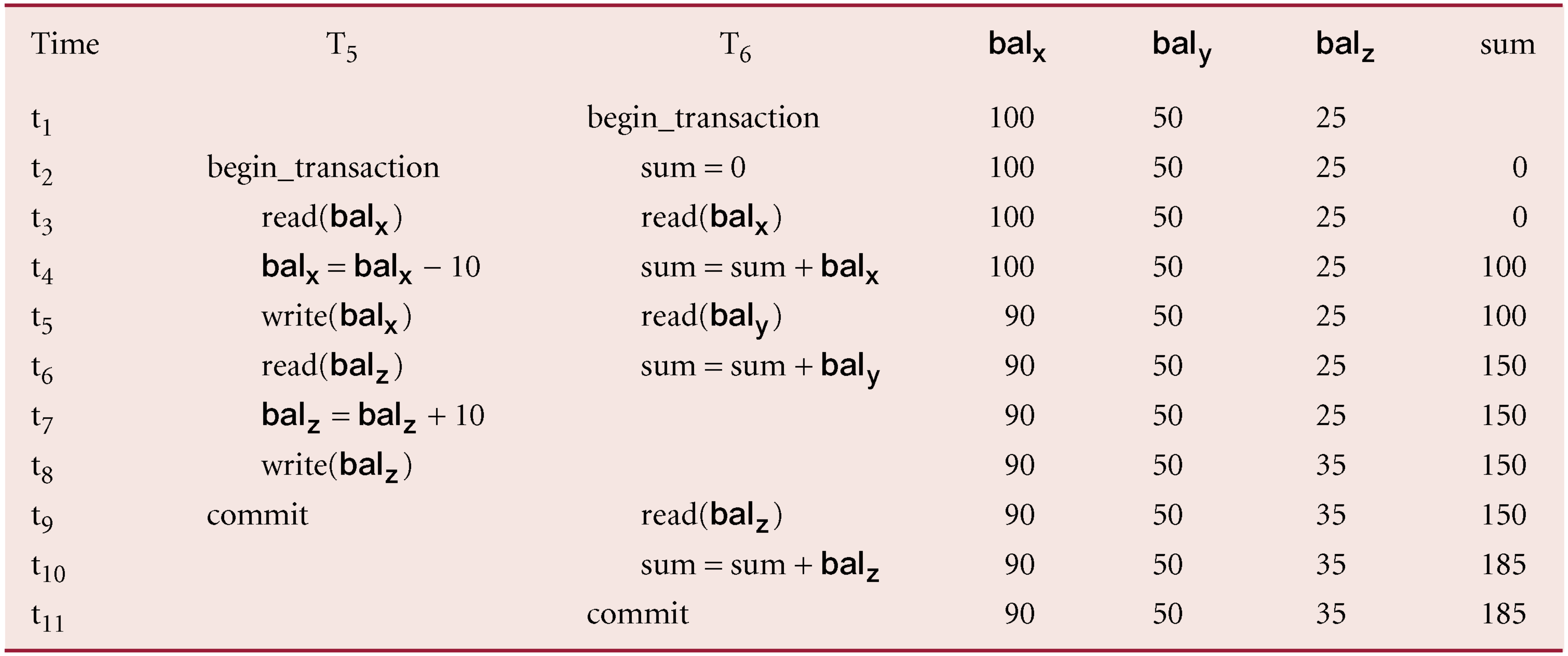
| Time | $T_3$ | $T_4$ | $bal_x$ |
|---|---|---|---|
| $t_1$ | | begin_transaction | 100 |
| $t_2$ | | read($bal_x$) | 100 |
| $t_3$ | | $bal_x = bal_x + 100$ | 100 |
| $t_4$ | begin_transaction | write($bal_x$) | 200 |
| $t_5$ | read($bal_x$) | ⋮ | 200 |
| $t_6$ | $bal_x = bal_x - 10$ | rollback | 100 |
| $t_7$ | write($bal_x$) | | 190 |
| $t_8$ | commit | | 190 |

## Inconsistent Analysis Problem

- An inconsistent analysis problem occurs when a transaction T reads several values but another transaction T′ updates some of them during the execution of T.
- This is sometimes referred to as an unrepeatable, or fuzzy read.
- For example:
    - T6 is totalling balances of account X (£100), account Y (£50), and account Z (£25).
    - Meanwhile, T5 has transferred £10 from balx to balz, so T6 now has the wrong result (£10 too high).

# Inconsistent Analysis Problem

- The problem is avoided by preventing T6 from reading balx and balz until after T5 completes the updates.



| Time | $T_5$ | $T_6$ | $bal_x$ | $bal_y$ | $bal_z$ | sum |
|---|---|---|---|---|---|---|
| $t_1$ | | begin_transaction | 100 | 50 | 25 | |
| $t_2$ | begin_transaction | sum = 0 | 100 | 50 | 25 | 0 |
| $t_3$ | read($bal_x$) | read($bal_x$) | 100 | 50 | 25 | 0 |
| $t_4$ | $bal_x = bal_x - 10$ | sum = sum + $bal_x$ | 100 | 50 | 25 | 100 |
| $t_5$ | write($bal_x$) | read($bal_y$) | 90 | 50 | 25 | 100 |
| $t_6$ | read($bal_z$) | sum = sum + $bal_y$ | 90 | 50 | 25 | 150 |
| $t_7$ | $bal_z = bal_z + 10$ | | 90 | 50 | 25 | 150 |
| $t_8$ | write($bal_z$) | | 90 | 50 | 35 | 150 |
| $t_9$ | commit | read($bal_z$) | 90 | 50 | 35 | 150 |
| $t_{10}$ | | sum = sum + $bal_z$ | 90 | 50 | 35 | 185 |
| $t_{11}$ | | commit | 90 | 50 | 35 | 185 |

## Serializability

- The goal of a concurrency control protocol is to schedule transactions in such a way as to avoid any interference.
- One could run transactions serially, but this would limit the degree of concurrency or parallelism in the system, thereby hitting performance.
- Serializability identifies those execution sequences of transactions that are guaranteed to ensure final consistency.

## Serializability

- A schedule is a sequence of reads/writes by a set of concurrent transactions.
- A serial schedule is a schedule where the operations of each transaction are executed consecutively without any interleaved operations from any other transaction.
- A non-serial schedule is a schedule where operations from any transaction in a set of concurrent transactions are interleaved.

## Serializability

- We do not aim to guarantee that the results of all serial executions of a given set of transactions will be identical.
- The goal of serializability is to find non-serial schedules that allow transactions to execute concurrently without interfering with one another.
- In other words, we want to find non-serial schedules that are equivalent to some serial schedule.
- Such schedules are said to be serializable.

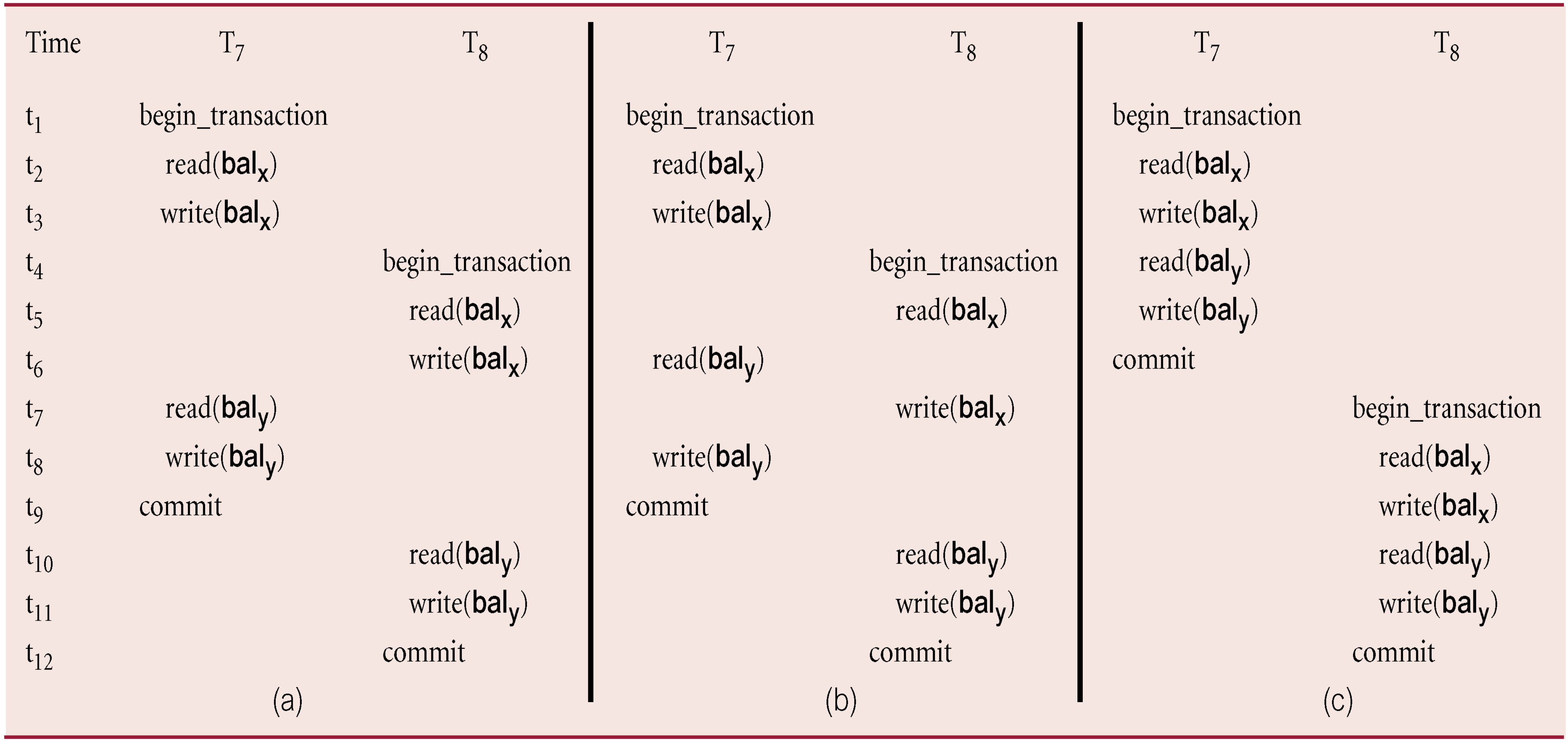
## Serializability

- In serializability, the ordering of reads/writes is important:

1. If two transactions only read a data item, they do not conflict and the order of execution is not important.
2. If two transactions either read or write completely separate data items, they do not conflict and the order of execution is not important.
3. If one transaction writes a data item and another reads or writes the same data item, the order of execution <u>is</u> important.

# Example of Conflict Serializability

**non-serial schedule S1** ≡ **non-serial schedule S2** ≡ **serial schedule S3**



| Time | $T_7$ | $T_8$ | $T_7$ | $T_8$ | $T_7$ | $T_8$ |
|---|---|---|---|---|---|---|
| $t_1$ | begin_transaction | | begin_transaction | | begin_transaction | |
| $t_2$ | read(**bal**$_x$) | | read(**bal**$_x$) | | read(**bal**$_x$) | |
| $t_3$ | write(**bal**$_x$) | | write(**bal**$_x$) | | write(**bal**$_x$) | |
| $t_4$ | | begin_transaction | | begin_transaction | read(**bal**$_y$) | |
| $t_5$ | | read(**bal**$_x$) | | read(**bal**$_x$) | write(**bal**$_y$) | |
| $t_6$ | | write(**bal**$_x$) | read(**bal**$_y$) | | commit | |
| $t_7$ | read(**bal**$_y$) | | | write(**bal**$_x$) | | begin_transaction |
| $t_8$ | write(**bal**$_y$) | | write(**bal**$_y$) | | | read(**bal**$_x$) |
| $t_9$ | commit | | commit | | | write(**bal**$_x$) |
| $t_{10}$ | | read(**bal**$_y$) | | read(**bal**$_y$) | | read(**bal**$_y$) |
| $t_{11}$ | | write(**bal**$_y$) | | write(**bal**$_y$) | | write(**bal**$_y$) |
| $t_{12}$ | | commit | | commit | | commit |
| | (a) | | (b) | | (c) | |

## Serializability

- A conflict-serializable schedule orders any conflicting operations in the same way as some serial execution.
- Under the constrained-write rule (i.e., a transaction must first read any data item it wants to update and does so based on its old value), we use a precedence graph to test for serializability.
- The opposite of a constrained write is a blind write, i.e., a transaction writes a value it has not read.
- Note that DBMSs do not test for schedule serializability (as the operating system will ultimately decide the interleaning of operations, not the DBMS).
- Instead, a DBMS will use protocols (see later) that are known to produce serializable schedules.

## Precedence Graph

- Create:
    - a node for each transaction; and then
    - a directed edge Ti → Tj, if
        - Tj reads the value of an item written by Ti; or
        - Tj writes a value into an item after it has been read by Ti; or
        - Tj writes a value into an item after it has been written by Ti.
- If the precedence graph contains a cycle, the schedule is not conflict-serializable.

## Example: Non-conflict-serializable schedule

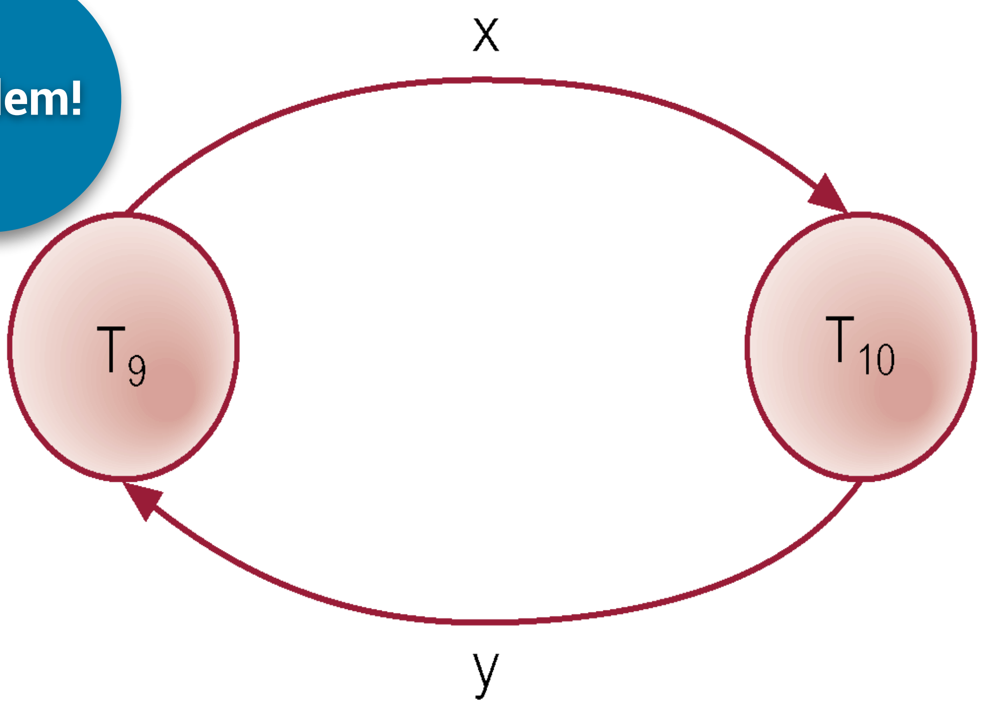
- T9 is transferring £100 from one account with balance balx to another account with balance baly.
- T10 is increasing balance of these two accounts by 10%.
- Precedence graph has a cycle and so is not serializable.

# Example: Non-conflict-serializable schedule

| Time | $T_9$ | $T_{10}$ |
|---|---|---|
| $t_1$ | begin_transaction | |
| $t_2$ | read($bal_x$) | |
| $t_3$ | $bal_x = bal_x + 100$ | |
| $t_4$ | write($bal_x$) | begin_transaction |
| $t_5$ | | read($bal_x$) |
| $t_6$ | | $bal_x = bal_x *1.1$ |
| $t_7$ | | write($bal_x$) |
| $t_8$ | | read($bal_y$) |
| $t_9$ | | $bal_y = bal_y *1.1$ |
| $t_{10}$ | | write($bal_y$) |
| $t_{11}$ | read($bal_y$) | commit |
| $t_{12}$ | $bal_y = bal_y - 100$ | |
| $t_{13}$ | write($bal_y$) | |
| $t_{14}$ | commit | |

problem!

problem!



$$\begin{tikzcd} T_9 \arrow[r, bend left, "x"] & T_{10} \arrow[l, bend left, "y"] \end{tikzcd}$$

## View Serializability

- View serializability offers a less stringent definition of schedule equivalence than conflict serializability.
- Some schedules are view-serializable but not all view-serializable schedules are conflict-serializable.
- Any view serializable schedule that is not conflict-serializable has one or more blind writes (i.e., some transaction writes a value that is has not previously read).
- Testing for view serializability is NP-complete, i.e., much harder than testing for conflict serializability.
- (For learning more about view serializability, see Chapter 22 in the Connolly and Begg textbook.)

## Recoverability

- Serializability identifies schedules that maintain database consistency assuming no transaction fails.
- One could also examine recoverability of transactions within the schedule.
- If a transaction fails, atomicity requires the effects of the transaction to be undone.
- Durability states that once a transaction commits, its changes cannot be undone (at least not without running another, compensating, transaction).

# Recoverable Schedule

- Consider the schedule on the left again.
- Assume that T9 rolls back rather than commit.
- But T10 has read (at t5) the outcome of the write by T9 at t4, used it and committed by the time that T9 rolls back.
- If T9 is to roll back, we should undo T10 because it used the outcome of a write that has been undone.
- But if we did that we would violate durability.
- The schedule on the left is therefore a nonrecoverable schedule.
- A recoverable schedule is one where, for each pair of transactions Ti and Tj, if Tj reads a data item previously written by Ti, then the commit operation of Ti precedes the commit operation of Tj.

| Time | $T_9$ | $T_{10}$ |
|---|---|---|
| $t_1$ | begin_transaction | |
| $t_2$ | read($bal_x$) | |
| $t_3$ | $bal_x = bal_x + 100$ | |
| $t_4$ | write($bal_x$) | begin_transaction |
| $t_5$ | | read($bal_x$) |
| $t_6$ | | $bal_x = bal_x * 1.1$ |
| $t_7$ | | write($bal_x$) |
| $t_8$ | | read($bal_y$) |
| $t_9$ | | $bal_y = bal_y * 1.1$ |
| $t_{10}$ | | write($bal_y$) |
| $t_{11}$ | read($bal_y$) | commit |
| $t_{12}$ | $bal_y = bal_y - 100$ | |
| $t_{13}$ | write($bal_y$) | |
| $t_{14}$ | commit | |

## Concurrency Control Techniques

- There are two basic concurrency control techniques:
  - Locking
  - Timestamping
- Both are pessimistic (or conservative) approaches, i.e., they delay a transaction in case it conflicts with other transactions.
- Optimistic methods assume that a conflict is rare, allows transactions to run and only check for conflicts at commit.

## Locking

- In locking, locks in a transaction are used to deny access to other transactions and so prevent incorrect updates.
- It is the most widely used approach to ensure serializability.
- Generally, a transaction must claim
  - a shared lock on a data item before the latter can be read and
  - an exclusive lock before it can be written.
- A lock prevents another transaction
  - from modifying the item if the lock is shared or
  - even from reading it if the lock is exclusive.

## Locking: Basic Rules

- If a transaction has a shared lock on an item, it can read but not update the item.
- If a transaction has an exclusive lock on an item, it can both read and update the item.
- Reads cannot conflict, so more than one transaction can hold shared locks simultaneously on the same item.
- An exclusive lock on an item gives a transaction exclusive access to that item.

## Locking: Basic Rules

- Some systems allow transactions to upgrade a shared lock to an exclusive lock, or downgrade an exclusive lock to a shared lock.
- Note that DBMSs also use another type of lock, called a latch, that is used mostly for disk transfers and therefore is normally held for shorter periods (more details on the mandatory readings).

## Example: Incorrect Locking Schedule

- Recall the example non-conflict-serializable schedule we saw before.
- Now, the schedule to the right uses the locking rules but it is still not serializable.
- If, when execution starts, balx =100 and baly = 400, then
  - the intended schedule would give balx = 220 and baly = 340
  - but T9 ; T10 gives balx = 220 and baly = 330
  - and T10; T9 gives balx = 210 and baly = 340
  - i.e., S is not serializable.

```
S = {
     write_lock(T9, balx),
     read(T9, balx),
     write(T9, balx+100),
     unlock(T9, balx),
                    write_lock(T10, balx),
                    read(T10, balx),
                    write(T10, balx*1.1),
                    unlock(T10, balx),
                    write_lock(T10, baly),
                    read(T10, baly),
                    write(T10, baly*1.1),
                    unlock(T10, baly),
                    commit(T10),
     write_lock(T9, baly),
     read(T9, baly),
     write(T9, baly-100),
     unlock(T9, baly),
     commit(T9)
     }
```

## Example: Incorrect Locking Schedule

- The problem is that the transactions are gaining/ releasing locks on an as-needed basis.
- The gain/release may happen too soon, resulting in loss of total isolation and atomicity.
- To guarantee serializability, we need an additional protocol concerning the positioning of lock and unlock operations in every transaction.
- The best known protocol for this is called two-phase locking (2PL).

## Two-Phase Locking (2PL)

- A transaction follows the 2PL protocol if all locking operations precede the first unlock operation in the transaction.
- There are two phases in each transaction:
  - Growing phase: here the transaction must acquire all the locks it needs but cannot release any locks.
  - Shrinking phase: here the transaction can release locks but cannot acquire any new locks.
- If upgrading/downgrading are supported then upgrading can only take place in the growing phase and downgrading can only take place in the shrinking phase.
- If every transaction in a schedule follows 2PL, the schedule is serializable.

## Preventing Lost Updates with 2PL

| Time | $T_1$ | $T_2$ | $bal_x$ |
|---|---|---|---|
| $t_1$ | | begin_transaction | 100 |
| $t_2$ | begin_transaction | write_lock($bal_x$) | 100 |
| $t_3$ | write_lock($bal_x$) | read($bal_x$) | 100 |
| $t_4$ | WAIT | $bal_x = bal_x + 100$ | 100 |
| $t_5$ | WAIT | write($bal_x$) | 200 |
| $t_6$ | WAIT | commit/unlock($bal_x$) | 200 |
| $t_7$ | read($bal_x$) | | 200 |
| $t_8$ | $bal_x = bal_x - 10$ | | 200 |
| $t_9$ | write($bal_x$) | | 190 |
| $t_{10}$ | commit/unlock($bal_x$) | | 190 |

# Preventing Uncommitted Dependencies w/ 2PL

| Time | $T_3$ | $T_4$ | $bal_x$ |
|---|---|---|---|
| $t_1$ | | begin_transaction | 100 |
| $t_2$ | | write_lock($bal_x$) | 100 |
| $t_3$ | | read($bal_x$) | 100 |
| $t_4$ | begin_transaction | $bal_x = bal_x + 100$ | 100 |
| $t_5$ | write_lock($bal_x$) | write($bal_x$) | 200 |
| $t_6$ | WAIT | rollback/unlock($bal_x$) | 100 |
| $t_7$ | read($bal_x$) | | 100 |
| $t_8$ | $bal_x = bal_x - 10$ | | 100 |
| $t_9$ | write($bal_x$) | | 90 |
| $t_{10}$ | commit/unlock($bal_x$) | | 90 |

# Preventing Inconsistent Analyses with 2PL

| Time | $T_5$ | $T_6$ | $bal_x$ | $bal_y$ | $bal_z$ | sum |
|---|---|---|---|---|---|---|
| $t_1$ | | begin_transaction | 100 | 50 | 25 | |
| $t_2$ | begin_transaction | sum = 0 | 100 | 50 | 25 | 0 |
| $t_3$ | write_lock($bal_x$) | | 100 | 50 | 25 | 0 |
| $t_4$ | read($bal_x$) | read_lock($bal_x$) | 100 | 50 | 25 | 0 |
| $t_5$ | $bal_x = bal_x - 10$ | WAIT | 100 | 50 | 25 | 0 |
| $t_6$ | write($bal_x$) | WAIT | 90 | 50 | 25 | 0 |
| $t_7$ | write_lock($bal_z$) | WAIT | 90 | 50 | 25 | 0 |
| $t_8$ | read($bal_z$) | WAIT | 90 | 50 | 25 | 0 |
| $t_9$ | $bal_z = bal_z + 10$ | WAIT | 90 | 50 | 25 | 0 |
| $t_{10}$ | write($bal_z$) | WAIT | 90 | 50 | 35 | 0 |
| $t_{11}$ | commit/unlock($bal_x$, $bal_z$) | WAIT | 90 | 50 | 35 | 0 |
| $t_{12}$ | | read($bal_x$) | 90 | 50 | 35 | 0 |
| $t_{13}$ | | sum = sum + $bal_x$ | 90 | 50 | 35 | 90 |
| $t_{14}$ | | read_lock($bal_y$) | 90 | 50 | 35 | 90 |
| $t_{15}$ | | read($bal_y$) | 90 | 50 | 35 | 90 |
| $t_{16}$ | | sum = sum + $bal_y$ | 90 | 50 | 35 | 140 |
| $t_{17}$ | | read_lock($bal_z$) | 90 | 50 | 35 | 140 |
| $t_{18}$ | | read($bal_z$) | 90 | 50 | 35 | 140 |
| $t_{19}$ | | sum = sum + $bal_z$ | 90 | 50 | 35 | 175 |
| $t_{20}$ | | commit/unlock($bal_x$, $bal_y$, $bal_z$) | 90 | 50 | 35 | 175 |

## Problems

- However, problems can still occur.
- One problem known as cascading rollback can occur depending on when locks are allowed to be released.
- Two other problems are:
  - deadlock: two transactions are held back together and neither can progress
  - livelock: without deadlock, a transaction is left waiting indefinitely to acquire a lock

# Cascading Rollback

| Time | $T_{14}$ | $T_{15}$ | $T_{16}$ |
|---|---|---|---|
| $t_1$ | begin_transaction | | |
| $t_2$ | write_lock($bal_x$) | | |
| $t_3$ | read($bal_x$) | | |
| $t_4$ | read_lock($bal_y$) | | |
| $t_5$ | read($bal_y$) | | |
| $t_6$ | $bal_x = bal_y + bal_x$ | | |
| $t_7$ | write($bal_x$) | | |
| $t_8$ | unlock($bal_x$) | begin_transaction | |
| $t_9$ | ⋮ | write_lock($bal_x$) | |
| $t_{10}$ | ⋮ | read($bal_x$) | |
| $t_{11}$ | ⋮ | $bal_x = bal_x + 100$ | |
| $t_{12}$ | ⋮ | write($bal_x$) | |
| $t_{13}$ | ⋮ | unlock($bal_x$) | |
| $t_{14}$ | ⋮ | ⋮ | |
| $t_{15}$ | rollback | ⋮ | |
| $t_{16}$ | | ⋮ | begin_transaction |
| $t_{17}$ | | ⋮ | read_lock($bal_x$) |
| $t_{18}$ | | rollback | ⋮ |
| $t_{19}$ | | | rollback |

## Cascading Rollback

- In the example, the transactions conform to 2PL.
    - Assume T14 aborts.
    - Since T15 is dependent on T14, T15 must also be rolled back.
    - Since T16 is dependent on T15, it too must be rolled back.
- This is called cascading rollback.
- To prevent this with 2PL, leave the release of all locks until end of transaction, which is referred to as rigorous 2PL.
- Alternatively, leave the release of exclusive locks until end of transaction, which is referred to as strict 2PL.

# Deadlock

An impasse that may result when two (or more) transactions are each waiting for locks held by the other to be released.

| Time | $T_{17}$ | $T_{18}$ |
|---|---|---|
| $t_1$ | begin_transaction | |
| $t_2$ | write_lock($\mathbf{bal_x}$) | begin_transaction |
| $t_3$ | read($\mathbf{bal_x}$) | write_lock($\mathbf{bal_y}$) |
| $t_4$ | $\mathbf{bal_x} = \mathbf{bal_x} - 10$ | read($\mathbf{bal_y}$) |
| $t_5$ | write($\mathbf{bal_x}$) | $\mathbf{bal_y} = \mathbf{bal_y} + 100$ |
| $t_6$ | write_lock($\mathbf{bal_y}$) | write($\mathbf{bal_y}$) |
| $t_7$ | WAIT | write_lock($\mathbf{bal_x}$) |
| $t_8$ | WAIT | WAIT |
| $t_9$ | WAIT | WAIT |
| $t_{10}$ | ⋮ | WAIT |
| $t_{11}$ | ⋮ | ⋮ |

## Deadlock

- There is only one way to break deadlock: abort one or more of the transactions.
- A deadlock should be transparent to users, so the DBMS should restart the transaction(s).
- There are three common strategies for handling deadlock:
  - Timeouts
  - Deadlock prevention
  - Deadlock detection and recovery

## Timeouts

- A transaction that requests lock will only wait for a system-defined period of time.
- If the lock has not been granted within this period, the lock request times out.
- In this case, the DBMS assumes the transaction is deadlocked (even though it may not be), aborts it and automatically restarts it.

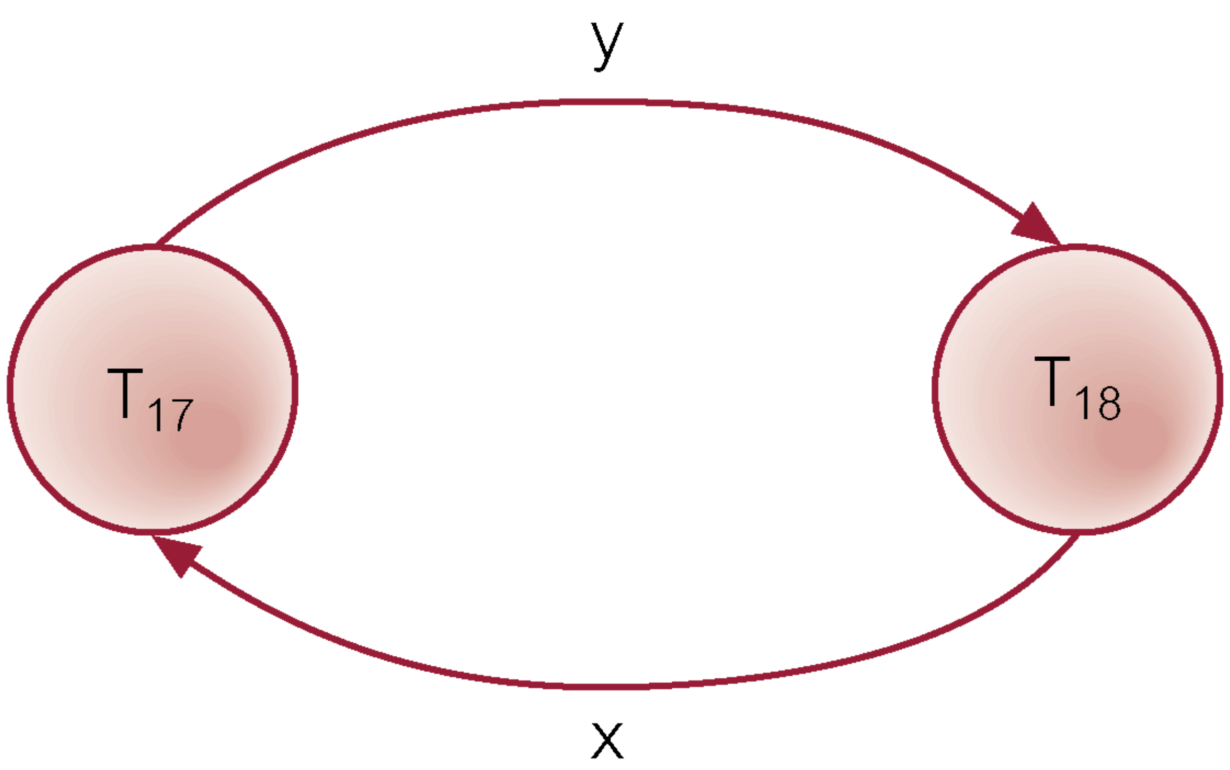
## Deadlock Prevention

- The DBMS looks ahead to see if the transaction would cause deadlock and never allows a deadlock to occur.
- We could order transactions using transaction timestamps:
  - Wait-Die: only an older transaction can wait for a younger one, otherwise the transaction is aborted (dies) and restarted with same timestamp (which means it will eventually become the older).
  - Wound-Wait: only a younger transaction can wait for an older one so that if an older transaction requests a lock held by a younger one, the younger one is aborted (wounded).
- A variant of 2PL called conservative 2PL can also be used.
  - It requires a transaction to acquire all locks it will need before it begins or else wait.
  - This is often onerous and hence not often deployed in practice.

## Deadlock Detection and Recovery

- The DBMS allows a deadlock to occur but recognizes it and breaks it.
- This is usually handled by the construction of a wait-for graph (WFG) showing transaction dependencies:
  - Create a node for each transaction.
  - Create an edge from Ti to Tj, if Ti is waiting to lock an item locked by Tj.
- Deadlock exists if and only if WFG contains a cycle.
- A WFG is created at regular intervals:
  - If this is done too frequently, the overhead may be too high.
  - If not frequently enough, the deadlock may persist for much too long.

## Example: Wait-For-Graph (WFG)



## Recovery from Deadlock Detection

- There are several issues:
  - the choice of deadlock victim (ideally the minimum re-run cost) will depend on:
    - how long it has been running
    - how many updates it has already done
    - how many updates it still has to do (which is difficult to estimate)
  - how far to roll a transaction back:
    - all the way to the start?
    - part of the way?
  - avoiding starvation:
    - if the same transaction is repeatedly chosen as victim, we come close to livelock as the affected transaction may find it hard to complete
    - we can keep a count of how many times it was chosen which, if reached, takes it out of contention for being aborted

## Timestamping

- Transactions can be ordered globally so that older-earlier transactions, i.e., transactions with smaller timestamps, get priority in the event of conflict.
- Conflict is resolved by rolling back and restarting the younger-later transaction.
- Since there are no locks, there is no deadlock.

## Timestamp

- A timestamp is a unique identifier created by the DBMS that indicates the relative starting time of a transaction.
- It can be generated using the system clock at the time the transaction started, or by simply incrementing a logical counter every time a new transaction starts.

## Timestamping

- Reads and writes proceed only if the last update on that data item was carried out by an older transaction.
- Otherwise, the transaction requesting the read or write is restarted and given a new timestamp.
- A new timestamp is needed to avoid repeated abort followed by restart.
- Data items are also given two timestamps:
  - read-timestamp: the timestamp of the last transaction to read item;
  - write-timestamp: the timestamp of the last transaction to write item.

## Timestamping

- A transaction T with timestamp ts(T) issues a read(x):
  - If ts(T) < write_timestamp(x), then x has already been updated by younger (later) transaction.
    - The values it may have acquired may be inconsistent with the updated value written by the later transaction.
    - T must be aborted and restarted with a new timestamp.
  - Otherwise, ts(T) >= write_timestamp(x), the read can proceed and we set
    - read_timestamp(X) := max(ts(T),read_timestamp(x))

# Timestamping

- A transaction T with timestamp ts(T) issues a write(x):
  - If ts(T) < read_timestamp(x), then x has already ben read by a younger (later) transaction.
    - The younger transaction may be taking decisions on the basis of the read value and it would be wrong for the older transaction to update it now.
    - T must be rolled back and restarted with a new timestamp.
  - If ts(T) < write_timestamp(x), then it has already ben updated by the younger transaction.
    - This means that T would write an obsolete value of x.
    - Either T must be rolled back and restarted with a new timestamp or we simply ignore the attempt by T to write.
    - (The latter is known as Thomas's write rule and allows greater concurrency.)
  - Otherwise, the write can proceed and we set
    - write_timestamp(X) := ts(T)

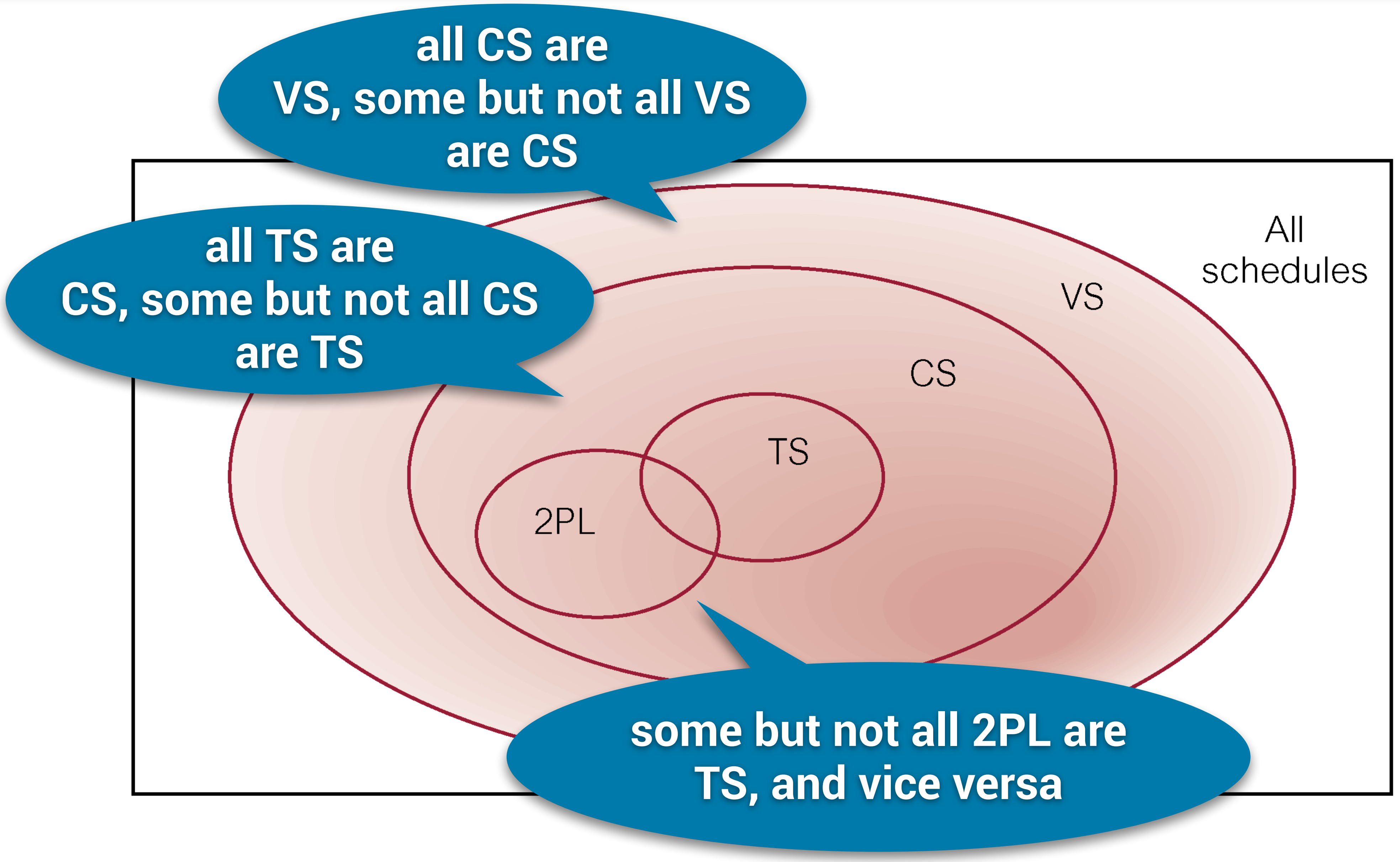
## Example – Basic Timestamp Ordering

| Time | Op | $T_{19}$ | $T_{20}$ | $T_{21}$ |
|---|---|---|---|---|
| $t_1$ | | begin_transaction | | |
| $t_2$ | read($bal_x$) | read($bal_x$) | | |
| $t_3$ | $bal_x = bal_x + 10$ | $bal_x = bal_x + 10$ | | |
| $t_4$ | write($bal_x$) | write($bal_x$) | begin_transaction | |
| $t_5$ | read($bal_y$) | | read($bal_y$) | |
| $t_6$ | $bal_y = bal_y + 20$ | | $bal_y = bal_y + 20$ | begin_transaction |
| $t_7$ | read($bal_y$) | | | read($bal_y$) |
| $t_8$ | write($bal_y$) | | write($bal_y$)$^+$ | |
| $t_9$ | $bal_y = bal_y + 30$ | | | $bal_y = bal_y + 30$ |
| $t_{10}$ | write($bal_y$) | | | write($bal_y$) |
| $t_{11}$ | $bal_z = 100$ | | | $bal_z = 100$ |
| $t_{12}$ | write($bal_z$) | | | write($bal_z$) |
| $t_{13}$ | $bal_z = 50$ | $bal_z = 50$ | | commit |
| $t_{14}$ | write($bal_z$) | write($bal_z$)$^‡$ | begin_transaction | |
| $t_{15}$ | read($bal_y$) | commit | read($bal_y$) | |
| $t_{16}$ | $bal_y = bal_y + 20$ | | $bal_y = bal_y + 20$ | |
| $t_{17}$ | write($bal_y$) | | write($bal_y$) | |
| $t_{18}$ | | | commit | |

$^+$ At time $t_8$, the write by transaction $T_{20}$ violates the first timestamping write rule described above and therefore is aborted and restarted at time $t_{14}$.

$^‡$ At time $t_{14}$, the write by transaction $T_{19}$ can safely be ignored using the ignore obsolete write rule, as it would have been overwritten by the write of transaction $T_{21}$ at time $t_{12}$.

# Comparison of Methods


all CS are VS, some but not all VS are CS
all TS are CS, some but not all CS are TS
All schedules
VS
CS
TS
2PL
some but not all 2PL are TS, and vice versa

VS = view serializability
CS = conflict-serializability
TS = timestamping
2PL = two-phase locking

## Multiversion Timestamp Ordering

- Versioning of data can be used to increase concurrency.
- The basic timestamp ordering protocol assumes that only one version of a data item exists, and so only one transaction can access the data item at a time.
- We can allow multiple transactions to read and write different versions of the same data item, and ensure that each transaction sees a consistent set of versions for all the data items it accesses.

## Multiversion Timestamp Ordering

- In multiversion concurrency control, each write operation creates a new version of data item while retaining old version.
- When a transaction attempts to read a data item, the system selects one version that ensures serializability.
- Versions can be deleted once they are no longer required.

## Optimistic Techniques

- Optimistic techniques are based on assumption that conflicts are rare and it is more efficient to let transactions proceed without delays to ensure serializability.
- At commit, a check is made to determine whether conflict has occurred.
- If there is a conflict, the transaction must be rolled back and restarted.
- This potentially allows greater concurrency than traditional protocols.

## Optimistic Techniques

- Three phases:
  - Read
    - It extends from start until immediately before commit.
    - A transaction reads values from database and stores them in local variables.
    - Updates are applied to a local copy of the data.
  - Validation (follows Read)
    - For a read-only transaction, it checks that values read are still current values.
    - If there was no interference, transaction is committed, else aborted and restarted.
    - For update transaction, it checks that the transaction leaves database in a consistent state, with serializability maintained.
  - Write (follows Validation of Update Transactions)
    - Updates made to local copy are applied to the database.

## Granularity of Data Items

- Granularity is the size of the data items chosen as unit of protection by a concurrency control protocol.
- It ranges from from coarse to fine:
  - An entire database
  - A file
  - A page (or area or database space)
  - A record
  - A field value of a record

## Granularity of Data Items

- There is a trade-off:
  - The coarser the granularity, the lower the degree of concurrency;
  - The finer the granularity, the more locking information that must be stored.
- The best item size depends on the type of transaction.

## Hierarchy of Granularity

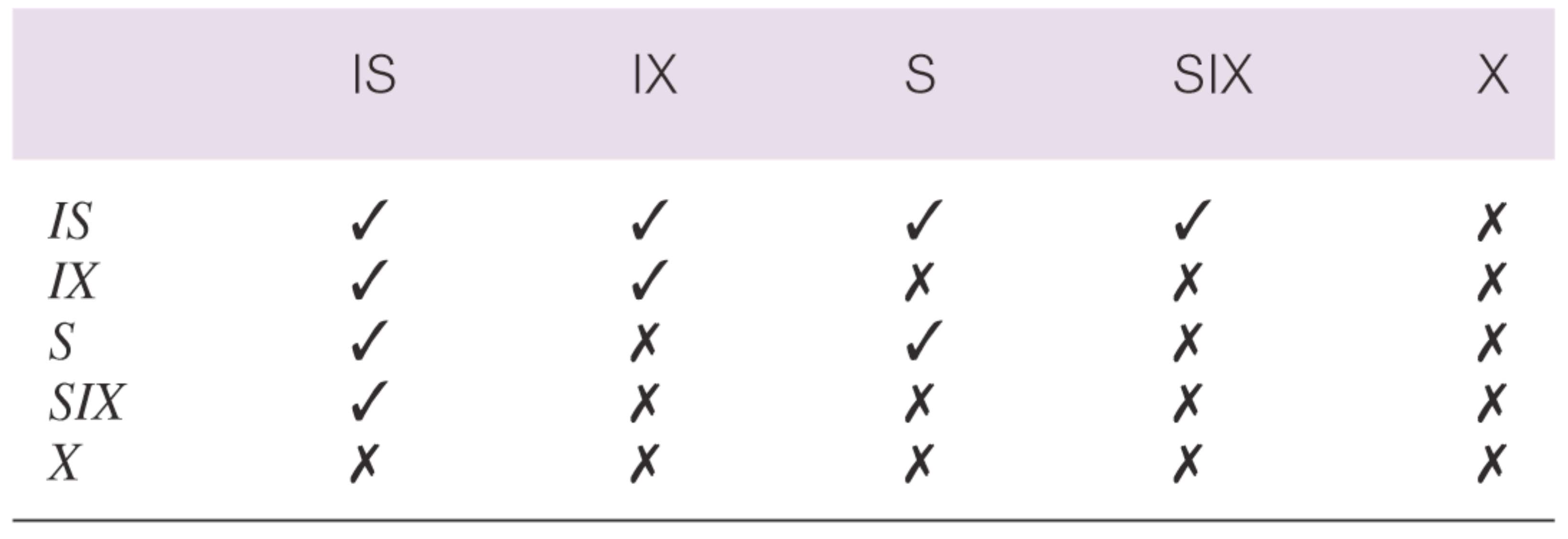
- One could represent the granularity of locks in a hierarchical structure.
- The root node represents an entire database, the children of the root node represent files, etc.
- When a node is locked, all its descendants are also locked.
- The DBMS therefore should check the hierarchical path before granting a lock.

# Hierarchy of Granularity

- An intention lock could be used to lock all ancestors of a locked node.
- Intention locks can be read or write.
- They are applied top-down, and released bottom-up.

| | |
|---|---|
| IS | = intention share |
| IX | = intention exclusive |
| S | = shared |
| SIX | = shared and intention exclusive |
| X | = exclusive |

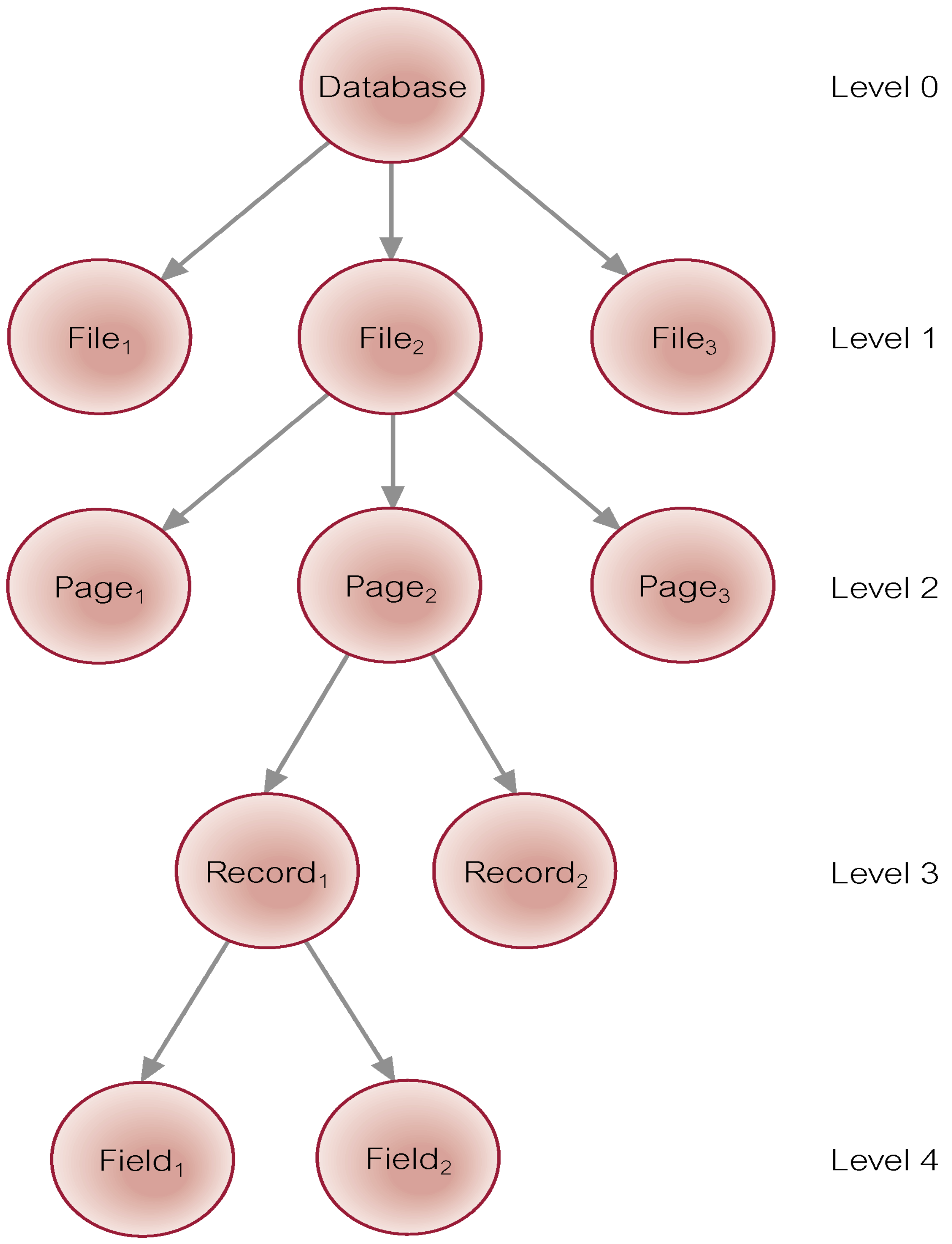
**Table 20.1** Lock compatibility table for multiple-granularity locking.



| | IS | IX | S | SIX | X |
|---|---|---|---|---|---|
| *IS* | ✓ | ✓ | ✓ | ✓ | ✗ |
| *IX* | ✓ | ✓ | ✗ | ✗ | ✗ |
| *S* | ✓ | ✗ | ✓ | ✗ | ✗ |
| *SIX* | ✓ | ✗ | ✗ | ✗ | ✗ |
| *X* | ✗ | ✗ | ✗ | ✗ | ✗ |

✓ = compatible, ✗ = incompatible

## Levels of Locking



## Isolation Levels

- Locks that are held until commit time, as is the case with strict protocols, have the effect of preventing more intense concurrency than less strict protocols.
- Because of this, most commercial DBMSs allow the application designer to choose among different locking protocols.
- They differ in what items they cause to be locked and for how long the locks are held.
- It is up to the designer to decide (from an understanding of the semantics of the application) which isolation level (i.e., which locking protocol) maximizes concurrency whilst preserving the correctness of the operations.
- Of course, other than the most stringent one, all other isolation levels allow for non-serialisable schedules.
- But, for a specific application and for some non-stringent isolation level, it may be the case that either non-serializability doesn't affect the correctness or else all the schedules produced at that isolation level are in fact serializable and therefore correct.

## Isolation Levels

- The ANSI-standard isolation levels are defined in terms of the anomalies that it prevents. An anomaly that is prevented at one level with strength S is alos prevented at every level stronger than S.

- In order of increasing strength, the ANSI-standard isolation levels are:

  - READ UNCOMMITTED, where dirty reads are possible

  - READ COMMITTED, where dirty reads are impossible but non-repeatable reads and phantoms are possible

  - REPEATABLE READ, where dirty reads and non-repeatable reads are impossible but phantoms are possible

  - SERIALIZABLE, where dirty reads, non-repeatable reads and phantoms are impossible and the schedule must be serialisable

  - Note that only the strongest offers strong protection by enforcing serializability.

  - There is more to locking and isolation then covered here. So, make sure you study the mandatory readings.

## Database Recovery

- Recovery is the process of restoring database to a correct state in the event of a failure.
- The need for recovery control arises because:
  - There are two types of storage: volatile (main memory) and nonvolatile.
  - Volatile storage does not survive system crashes.
  - Stable storage represents information that has been replicated in several nonvolatile storage media with independent failure modes.

## Types of Failures

- System crashes: resulting in loss of main memory.
- Media failures: resulting in loss of parts of secondary storage.
- Application software errors.
- Natural physical disasters.
- Carelessness or unintentional destruction of data or facilities.
- Sabotage.

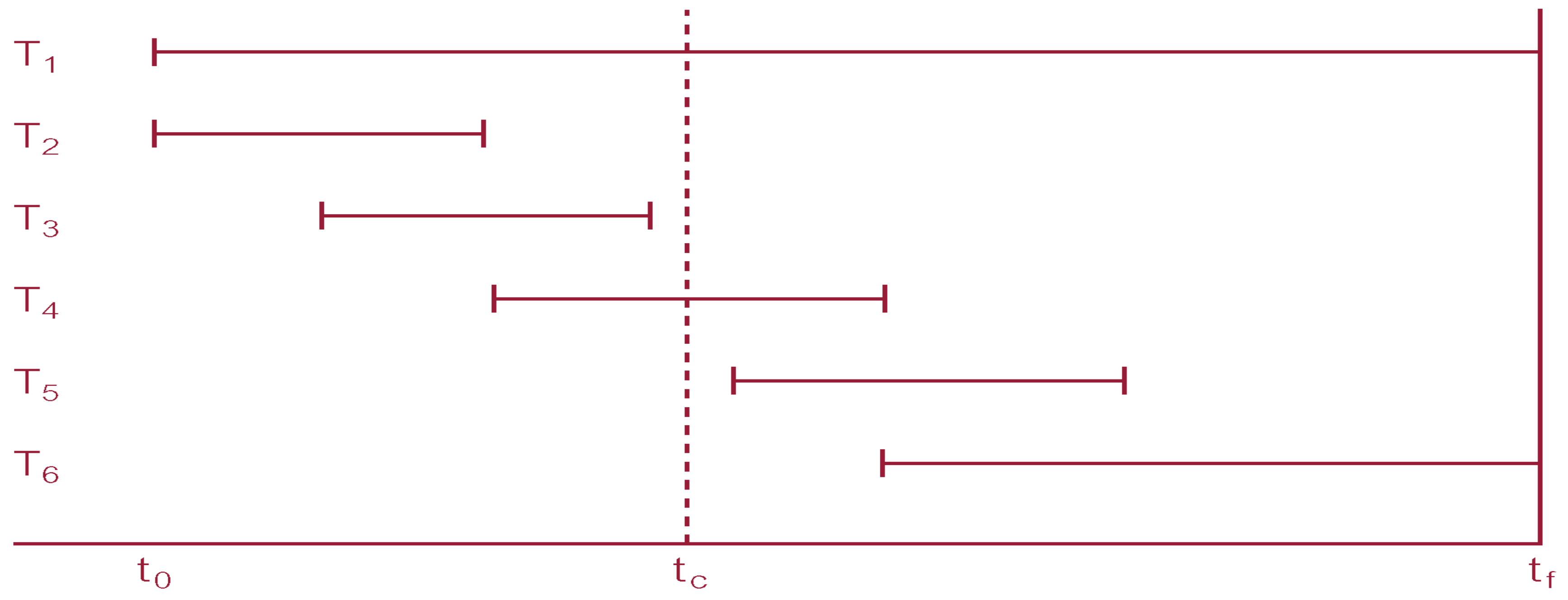
## Transactions and Recovery

- Transactions represent the basic unit of recovery.
- The recovery manager is responsible for atomicity and durability.
- If a failure occurs between a commit and the database buffers being flushed to secondary storage, then, to ensure durability, the recovery manager has to redo (roll-forward) the updates made by the transaction.

## Transactions and Recovery

- If the transaction had not committed at failure time, the recovery manager has to undo (roll-back) any effects of that transaction to ensure atomicity.
- Undos can be partial or global:
  - Partial undo: only one transaction has to be undone.
  - Global undo: all transactions have to be undone.

## Example



- The DBMS starts at time t0, but fails at time tf. Assume the data for transactions T2 and T3 have been written to secondary storage.
- T1 and T6 have to be undone.
- In the absence of any other information, recovery manager has to redo T2, T3, T4, and T5.

## Recovery Facilities

- DBMS should provide following facilities to assist with recovery:
- Backup mechanism, which makes periodic backup copies of database.
  - Logging facilities, which keep track of current state of transactions and database changes.
  - Checkpoint facility, which enables updates to database in progress to be made permanent.
  - Recovery manager, which allows DBMS to restore database to consistent state following a failure.

## Log File

- The log file contains information about all updates to database:
  - Transaction records.
  - Checkpoint records.
- It is also often used for other purposes (for example, auditing).

## Log File

- The transaction records contain:
  - A transaction identifier.
  - The type of log record: transaction start, insert, update, delete, abort, commit.
  - An identifier of the data item affected by database action (i.e., insert, delete, and update operations).
  - The 'before' image of the data item.
  - The 'after' image of the data item.
  - Log management information.

# Sample Log File

| Tid | Time | Operation | Object | Before image | After image | pPtr | nPtr |
|---|---|---|---|---|---|---|---|
| T1 | 10:12 | START | | | | 0 | 2 |
| T1 | 10:13 | UPDATE | STAFF SL21 | (old value) | (new value) | 1 | 8 |
| T2 | 10:14 | START | | | | 0 | 4 |
| T2 | 10:16 | INSERT | STAFF SG37 | | (new value) | 3 | 5 |
| T2 | 10:17 | DELETE | STAFF SA9 | (old value) | | 4 | 6 |
| T2 | 10:17 | UPDATE | PROPERTY PG16 | (old value) | (new value) | 5 | 9 |
| T3 | 10:18 | START | | | | 0 | 11 |
| T1 | 10:18 | COMMIT | | | | 2 | 0 |
| | 10:19 | CHECKPOINT | T2, T3 | | | | |
| T2 | 10:19 | COMMIT | | | | 6 | 0 |
| T3 | 10:20 | INSERT | PROPERTY PG4 | | (new value) | 7 | 12 |
| T3 | 10:21 | COMMIT | | | | 11 | 0 |

## Log File

- A log file may be duplexed or triplexed.
- A log file is sometimes split into two separate random-access files.
- It is a potential bottleneck and hence critical in determining overall performance.

## Checkpointing

- A checkpoint is a point of synchronization between the database and the log file when all buffers are force-written to secondary storage.
- A checkpoint record is created containing the identifiers of all active transactions.
- When a failure occurs, we redo all the transactions that have committed since the checkpoint and undo all the transactions active at the time of the crash.

## Checkpointing

- In the previous example, with a checkpoint at time tc, the changes made by T2 and T3 have been written to secondary storage.
- Thus:
  - we only redo T4 and T5,
  - we undo transactions T1 and T6.

## Recovery Techniques

- If the database has been damaged:
  - We need to restore last backup copy of database and reapply the updates of committed transactions using the log file.
- If the database is only inconsistent:
  - We need to undo the changes that caused inconsistency.
  - We may also need to redo some transactions to ensure the updates reach secondary storage.
  - We do not need a backup if we can restore database using the before- and after-images in the log file.

## Main Recovery Techniques

- There are three main recovery techniques:
  - Deferred Update
  - Immediate Update
  - Shadow Paging

## Deferred Update

- In this case, updates are not written to the database until after a transaction has reached its commit point.
- If a transaction fails before commit, it will not have modified database and so no undoing of changes required.
- It may be necessary to redo updates of committed transactions as their effect may not have reached database.

## Immediate Update

- In this case, updates are applied to database as they occur.
- We need to redo the updates of committed transactions following a failure.
- We may need to undo the effects of transactions that had not committed at the time of failure.
- It is essential that log records are written before we write to the database, i.e., we must use a write-ahead log protocol.
- If no “transaction commit” record in log, then that transaction was active at failure time and must be undone.
- Undo operations are performed in reverse order to that in which they were written to the log.

## Shadow Paging

- In this case, we maintain two page tables during life of a transaction:
  - a current page table and
  - a shadow page table.
- When a transaction starts, the two pages are the same.
- The shadow page table is never changed thereafter and is used to restore the database in event of failure.
- During the transaction, the current page table records all updates to database.
- When the transaction completes, the current page table becomes the shadow page table.

## Advanced Transaction Models

- The protocols considered so far are suitable for types of transactions that arise in traditional business applications, in which:
  - Data items are simple and small
  - Transactions have short duration and involve few data items
- In constrast, in other kinds of application, like computer-aided design:
  - Data items have many types, sometimes forming large collections.
  - Designs may be very large in terms of the data items used.
  - Designs are not static but evolve through time.
  - Updates can be far-reaching, involving many data items.
  - Many people cooperate in designs, and work in parallel on them.

## Advanced Transaction Models

- These characteristics may result in transactions of long duration, giving rise to the following problems:
  - They are more susceptible to failure: so we need to minimize amount of work lost.
  - They may access a large number of data items: concurrency becomes limited if many data items are inaccessible for long periods.
  - Deadlock becomes more likely.
  - Cooperation through the use of shared data items is largely restricted by traditional concurrency protocols.
- This has led to the development of advanced transaction models (see the Connoly and Begg text if you want to learn more).

## In The Next Handout

- We'll explore what types of file organization there are, what indices are and how they play an important role in DBMSs.