Transaction Management

ECE 356
University of Waterloo
Dr. Paul A.S. Ward

Acknowledgment: slides derived from Dr. Wojciech Golab

based on materials provided by
Silberschatz, Korth, and Sudarshan, copyright 2010
(source: www.db-book.com)

Learning Outcomes

- Transactions
  - ACID, state, schedules, serializability, recoverability, isolation levels
  - Concurrency control:
    - locking modes, compatibility, 2PL, acquisition, granularity, manager
    - multiversion two-phase locking, timestamp ordering protocol
    - deadlock handling
  - Database recovery:
    - failure classification, log-based recovery, checkpoints
    - recovery from failure with loss of non-volatile storage

- Textbook sections (6th ed.): chapters 14,15, 16

Transaction Concept

- A **transaction** is a *atomic (indivisible) unit* of program execution that accesses and possibly updates various data items.
- Example: transaction to transfer $50 from account A to account B:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B*)
- Two important issues must be dealt with in transaction processing:
  - Failures of various kinds, such as hardware failures and system crashes.
  - Concurrent execution of multiple transactions.

ACID Properties: Example

- Transaction to transfer $50 from account *A* to account *B*:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B)*
- **Atomicity**
  - if $50 is debited from *A*, then it is deposited in *B* or credited back to *A*
- **Consistency**
  - the total balance of *A* plus *B* is preserved
- **Durability**
  - if $50 is deposited in *B*, it will not disappear from *B*
- **Isolation**
  - two transfers executing in parallel should be unaware of each other

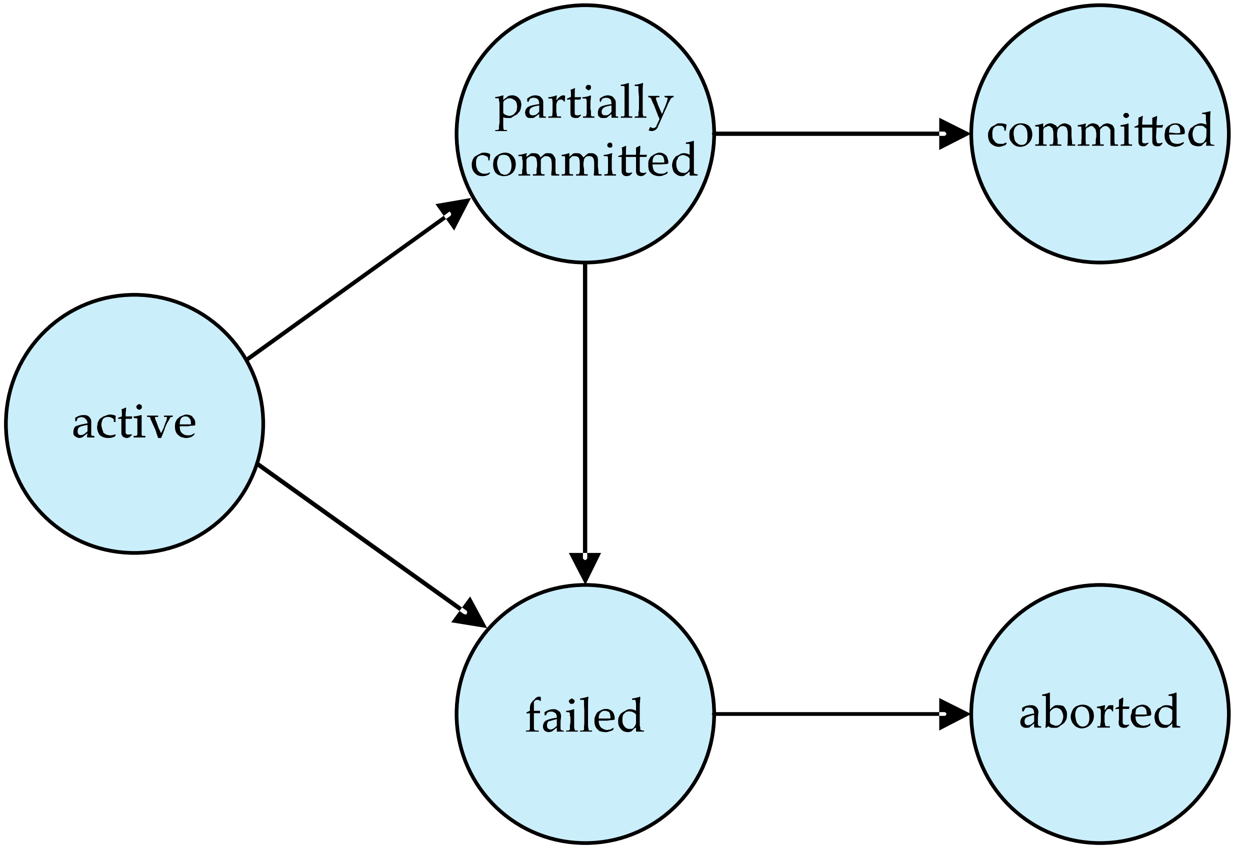
ACID Properties: Informal Definition

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
  (This includes integrity constraints, check constraints, and assertions.)
- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
- **Durability.** After a transaction completes successfully (*i.e.*, without aborting and rolling back), the changes it has made to the database persist, even if failures occur.

Transaction State

- **Active**: The initial state. The transaction remains in this state while it is executing.
- **Partially committed**: After the final statement has been executed, before updates are reflected in the database.
- **Failed**: After discovering that normal execution can no longer proceed, and hence the transaction must be rolled back.
- **Aborted**: After the transaction has been rolled back and the database is restored to its state prior to the start of the transaction. Two options after an abort:
  1. Restart the transaction.
     (*E.g.*, if it aborted due to a crash failure or due to contention.)
  2. Kill the transaction.
     (*E.g.*, if it aborted due to a logic error that must be debugged.)
- **Committed**: After successful completion, updates are reflected in the database.

Transaction State (Cont.)



Concurrent Executions

- **Concurrency:** when multiple transactions run simultaneously.
- Concurrency enables better performance:
  - **Increased processor and disk utilization**, leading to better transaction *throughput*. One transaction can be using the CPU while another is reading from or writing to the disk
  - **Reduced average response time** for transactions: short transactions need not wait behind long ones.
- Concurrency complicates the internal design of the database. A **concurrency control mechanism** is needed to preserve ACID properties, for example when two transactions attempt to access the same row of the same table.

Schedules

- Motivation: To define concepts such as serializable isolation in a precise manner we need a rigorous framework for reasoning about concurrent execution of transactions.
- A **schedule** is a sequence of instructions that specify the chronological order in which instructions of concurrent transactions are executed.
- A **serial schedule** is one in which a transaction that has been started runs to completion before another transaction may start.
- A transaction that successfully completes its execution ends with a **commit instruction**.
- A transaction that fails to successfully complete its execution ends with an **abort instruction**.
- The commit and abort instructions are sometimes omitted for brevity when the outcome of the transaction is clear from the context.
- Note: Our simplified model of transactions only captures read and write operations on data objects (*e.g.*, SQL select statements). It does not take into account the creation or deletion of such objects.

Example: Schedule 1

- Let T_1 transfer \$50 from *A* to *B*, and T_2 transfer 10% of the balance from *A* to *B.*
- Example: a serial schedule in which T_2 follows T_1.

T_1	T_2
read (*A*)	
$A := A - 50$	
write (*A*)	
read (*B*)	
$B := B + 50$	
write (*B*)	
commit	
	read (*A*)
	$temp := A * 0.1$
	$A := A - temp$
	write (*A*)
	read (*B*)
	$B := B + temp$
	write (*B*)
	commit

Example: Schedule 2

- A serial schedule in which T_1 follows T_2:

T_1	T_2
	read (A)
	$temp$:= A * 0.1
	A := A - $temp$
	write (A)
	read (B)
	B := B + $temp$
	write (B)
	commit
read (A)	
A := A – 50	
write (A)	
read (B)	
B := B + 50	
write (B)	
commit	

Example: Schedule 3

- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is ***equivalent*** (defined more precisely later on) to Schedule 1.

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (B)
	$B := B + temp$
	write (B)
	commit

In Schedules 1, 2 and 3, the sum A + B is preserved.

Example: Schedule 4

- The following schedule does not preserve (*A* + *B*), and is therefore not equivalent to Schedule 1.

T_1	T_2
read (*A*)	
A := *A* – 50	
	read (*A*)
	temp := *A* * 0.1
	A := *A* - *temp*
	write (*A*)
	read (*B*)
write (*A*)	
read (*B*)	
B := *B* + 50	
write (*B*)	
commit	
	B := *B* + *temp*
	write (*B*)
	commit

Serializability

- **Assumption:** Each transaction, when executed in isolation, preserves database consistency (*e.g.*, integrity constraints).
- This assumption implies that any serial schedule containing possibly many transactions also preserves database consistency.
- Informally speaking, a schedule is **serializable** if it is *equivalent* to a serial schedule. As a result, a serializable schedule also preserves database consistency even though the transactions in that schedule *may not be executed serially*.
- Different definitions of schedule equivalence give rise to the formal notions of **conflict serializability** and **view serializability**.

Conflicting Instructions

- Instructions I_i and I_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both I_i and I_j, and at least one of these instructions wrote Q.
  1. I_i = **read**(Q), I_j = **read**(Q). I_i and I_j do not conflict.
  2. I_i = **read**(Q), I_j = **write**(Q). They do conflict.
  3. I_i = **write**(Q), I_j = **read**(Q). They do conflict
  4. I_i = **write**(Q), I_j = **write**(Q). They do conflict
- Intuitively, a conflict between I_i and I_j forces a logical ordering between them.
- If I_i and I_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule. (The two instructions commute!)

Conflict Serializability

- **Definition:**
  Let *S* and *S′* be schedules for some set *R* of transactions.
  If schedule *S* can be transformed into schedule *S′* by a series of swaps of non-conflicting instructions, then we say that *S* and *S′* are **conflict equivalent***.*

  Note: None of the swaps can change the order of instructions that belong to the same transaction. This follows from the definition of *S* and *S′* as schedules over the same set of transactions.

- We say that a schedule *S* is **conflict serializable** if it is *conflict equivalent to a serial schedule*.

Conflict Serializability (2)

- Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions.
- Therefore, Schedule 3 is conflict serializable.

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

Schedule 3

T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

Schedule 6

Conflict Serializability (3)

- Example of a schedule that is not conflict serializable:

T_3	T_4
read (Q)	
	write (Q)
write (Q)	

- We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >. Why?

View Serializability

- Let *S* and *S′* be two schedules with the same set of transactions. *S* and *S′* are **view equivalent** if the following three conditions are met for each data item *Q*:
  1. If in schedule S a **read**(*Q*) of transaction T_i returns the initial value of *Q*, then in schedule *S′* the corresponding **read**(*Q*) of T_i must also return the initial value of *Q.*
  2. If in schedule S, a **read**(*Q*) of transaction T_i returns the value written by some **write**(Q) of transaction T_j (if any), then in schedule *S′* the corresponding **read**(*Q*) of T_i must also return the value written by the corresponding **write**(Q) of transaction T_j .
  3. The transaction (if any) that performs the final **write**(*Q*) operation in schedule *S* must also perform the final **write**(*Q*) operation in schedule *S′.*

View Serializability (2)

- A schedule *S* is **view serializable** if it is view equivalent to a serial schedule.
- **Theorem:** Every conflict serializable schedule is also view serializable.
  - The converse is *not* true
- Below is a schedule that is view-serializable but *not* conflict serializable:

T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

- What serial schedule is above view-equivalent to and why?

Testing for Conflict Serializability

- Consider a given schedule of a set of transactions T_1, T_2, ..., T_n
- **Precedence graph:** a directed graph where the vertices are the transactions (denoted by their names), and edges represent conflicting operations.
- We draw an edge from T_i to T_j if the two transactions contain conflicting instructions on some data item X, and:
  - T_i does a **write**(X) before T_j does a **read**(X), or
  - T_i does a **read**(X) before T_j does a **write**(X), or
  - T_i does a **write**(X) before T_j does a **write**(X)

- **Observation:** An edge from T_i to T_j implies that T_i must precede T_j in any conflict-equivalent serial schedule (if one exists at all).

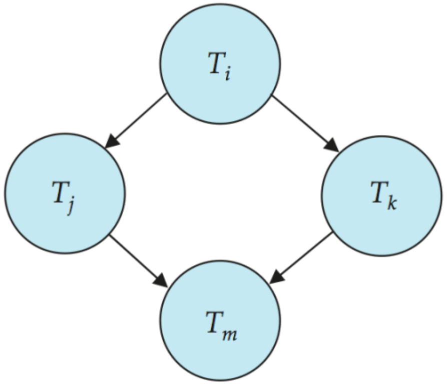
Test for Conflict Serializability

- **Theorem:** A schedule is conflict serializable if and only if its precedence graph is acyclic.
- Cycle-detection can be done in *O*(*n* + *e*) time where *n* is the number of vertices in the graph and *e* is the number of edges.
- If the precedence graph is acyclic, the order of transactions in an equivalent serial schedule can be obtained by a *topological sorting* of the graph.
- In the diagram on the right:

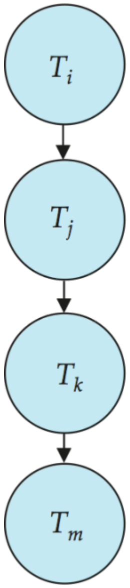
  (a) is the precedence graph

  (b) is one conflict-equivalent serial schedule

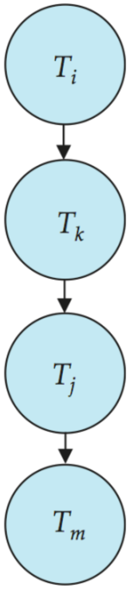
  (c) is another such schedule



(a)



(b)



(c)

Test for View Serializability

- The precedence graph test for conflict serializability cannot be used directly to test for view serializability.
  - If the precedence graph is acyclic then the schedule is view-serializable, but if there is a cycle in the graph then the schedule may or may not be view-serializable.
  - Extension to test for view serializability has cost exponential in the size of the precedence graph!
- The problem of checking if a schedule is view serializable falls in the class of *NP*-complete problems. Thus, the existence of an efficient algorithm is *extremely* unlikely.
- Practical algorithms that just check some **sufficient conditions** for view serializability can still be used, but are prone to false negatives.
  - Is a false negative bad?
  - What would happen if they were prone to false positives?

Recoverable Schedules

- Motivation: we must address the effect of transaction failures on concurrently executing transactions.
- **Recoverable schedule:** if a transaction T_j reads a data item previously written by a transaction T_i, and if T_j later commits successfully, then T_i also commits successfully and moreover the commit operation of T_i occurs before the commit operation of T_j.
- The following schedule (Schedule 11) is not recoverable because T_9 commits immediately after the read of A:

T_8	T_9
read (A)	
write (A)	
	read (A)
	commit
read (B)	

- If T_8 should abort, T_9 would have read (and possibly returned to the user) an inconsistent database state.

Cascading Rollbacks

- **Cascading rollback:** a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable):

T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		

If T_{10} fails, T_{11} and T_{12} must also be rolled back.

- Can lead to the undoing of a significant amount of work, hence wasted CPU cycles and disk IOs.

Cascadeless Schedules

- **Cascadeless schedules:** cascading rollbacks cannot occur. That is, for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, transaction T_i commits successfully and moreover the commit operation of T_i occurs before the read operation of T_j.
- For performance it is desirable to restrict the schedules to those that are cascadeless.
- **Theorem:** Every cascadeless schedule is a recoverable schedule.

- Are serial schedules recoverable? Are they cascadeless?

Weaker-than-Serializable Isolation

- Some applications can tolerate weak levels of transaction isolation:
  - Example 1: a read-only transaction that only wants to get an approximate total balance of all accounts.
  - Example 2: approximate database statistics computed for query optimization.
- The ability to choose among multiple isolation levels make it possible to trade off correctness (*e.g.*, accuracy of statistics) for performance.

Transaction Isolation Anomalies

- **Phantom read:** The results of a query in one transaction are changed by another transaction before the former commits.
  - Example: T1 does a SELECT COUNT(*) on a table, then T2 inserts a record into the same table before T1 has committed.
- **Non-repeatable read:** Repeated reads of same record in one transaction return different values because of an update made by another transaction.
  - Example: T1 reads a record, then T2 updates the same record, then T1 reads the record again and sees the updated value. **(Not a conflict-serializable schedule!)**
- **Dirty read:** One transaction reads a value written by another transaction that has not yet committed.
  - Example: T1 updates a record in a table, then T2 reads the updated record before T1 has committed. **(Not a cascadeless schedule, not recoverable either if T2 commits before T1!)**

Transaction Isolation Levels in SQL-92

Isolation Level	Allows phantom reads?	Allows non-repeatable reads?	Allows dirty reads?
serializable	no	no	no
repeatable read	yes	no	no
read committed	yes	yes	no
read uncommitted	yes	yes	yes

Note: The InnoDB storage engine in MySQL provides **repeatable read** by default.

Food for thought: Which SQL-92 isolation level corresponds to our definition of conflict-serializability? How about view-serializability?

Transactions in JDBC

```
Connection conn = ...
try {
    conn.setAutoCommit(false);
    conn.setTransactionIsolation(
        Connection.TRANSACTION_SERIALIZABLE);
    Statement stmt = conn.createStatement();
    String SQL = "INSERT INTO Pet " +
    "VALUES (106, 'Ace', 5, 'dog', 'beagle')";
    stmt.executeUpdate(SQL);
    String SQL = "INSERT INTO Owns VALUES (106, 110)";
    stmt.executeUpdate(SQL);
    conn.commit();
} catch(SQLException se){
    conn.rollback();
}
```

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are:
  - conflict- or view-serializable, and
  - are recoverable and preferably cascadeless
    - Why cascadeless?
- A policy in which only one transaction can execute at a time generates only serial schedules, but severely restricts concurrency.
  - Why do we want concurrency?
- Testing a schedule for serializability *after* it has executed is too late! The database must "think on its feet" instead.
- **Goal:** Develop concurrency control protocols that will guarantee serializability by design.
- **Observation:** Concurrency control protocols generally do not execute serializability tests internally. However, we can use serializability tests to understand why a concurrency control protocol is correct.

Concurrency Control

- A database must ensure ACID properties despite concurrent execution of transactions and the possibility of failures.
- **Concurrency control** is the mechanism that provides transaction isolation. Its purpose is to avoid isolation anomalies:
  - phantom reads
  - non-repeatable reads
  - dirty reads
- Serializable isolation prevents all three types of anomalies. Weaker isolation levels are also defined in SQL-92:
  - repeatable read
  - read committed
  - read uncommitted

Pessimistic vs. Optimistic Protocols

- There are two paradigms for designing concurrency control protocols:

  Pessimistic: check for synchronization conflicts during transaction execution, and if a conflict is detected then delay a data operation or abort an entire transaction.

  - Avoids wasted work when conflicts occur frequently.
  - Example 1: two-phase locking protocol (delay on conflict).
  - Example 2: timestamp ordering protocol (abort on conflict).

  Optimistic: allow transactions to proceed unchecked and detect synchronization conflicts when a transaction attempts to commit.

  - Reduces overhead when conflicts occur infrequently.
  - Example: backward validation protocol.

- **Analogy (Pessimistic):** The driver of a car either checks their blind spot before changing lanes to avoid a collision (pessimistic), or changes lanes without looking and worries about a collision after the fact (optimistic).
- **Analogy (Optimistic):** The entrepreneur starts a company and adjusts the product and sales strategy according to market reaction (optimistic) rather than doing a detailed market survey first (pessimistic).

Lock-Based Protocols

- A **lock** is a mechanism for restricting concurrent access to a data item (*e.g.*, table row, range of rows, entire table, etc.).
- For now, we assume that data items are table rows and can be locked in only two modes*:

1. **exclusive** (X) mode.
    - Transaction that owns the lock can read or write the data item.
    - X-lock is requested using the **lock-X** instruction.
2. **shared** (S) mode.
    - Transaction that owns the lock can only read the data item
    - S-lock is requested using the **lock-S** instruction.

- Transactions request locks from a *lock manager*.
- The lock manager may force the transaction to wait until the lock is released, or it may decide to abort the transaction (*e.g.*, to avoid deadlock).

* Oracle has 7 locking modes across three types, DB2 has 10 locking modes, ...

Lock-Based Protocols (Cont.)

- **Lock-compatibility matrix** for shared and exclusive locks:

	S	X
S	true	false
X	false	false

- **Explanation:** Imagine that two transactions request locks on the same data item. The compatibility matrix tells us which combinations of locks can be granted at the same time (denoted "true"), and which cannot (denoted "false").
- A transaction may be granted a lock on an item if the requested lock is compatible with all locks already held on that item by other transactions.
  - Any number of transactions can hold S-locks on an item.
  - If any transaction holds an X-lock on the item, then no other transaction may hold any lock on the item.

Lock-Based Protocols (Cont.)

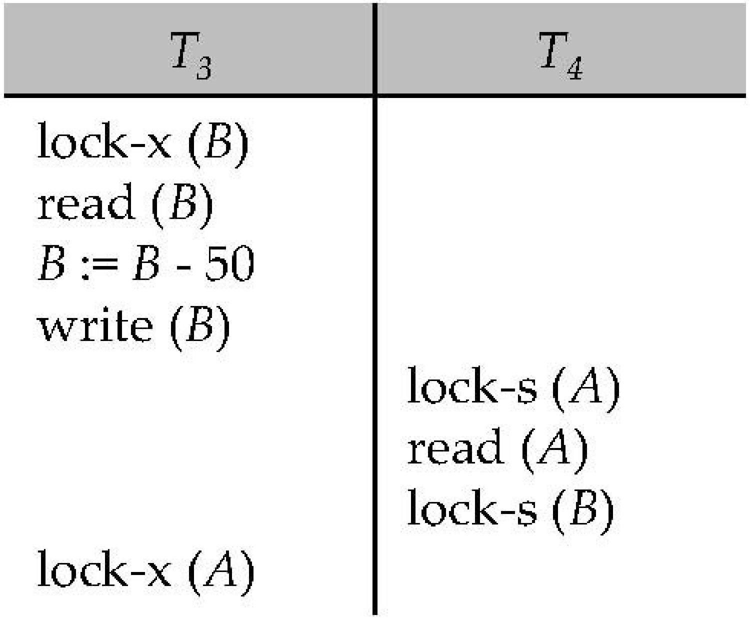
- Example of a transaction performing locking:

  T_2: **lock-S**(*A*)
  read(*A*)
  unlock(*A*)
  lock-S(*B*)
  read(*B*)
  unlock(*B*)
  display(*A*+*B*)

- **Note:** Locking as shown above *does not guarantee serializability!* If *A* or *B* is updated in-between the read of *A* and the read of *B*, then the displayed sum would be incorrect.
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

Pitfalls of Lock-Based Protocols

- Consider the partial schedule:



T_3	T_4
lock-x (B)	
read (B)	
$B := B - 50$	
write (B)	
	lock-s (A)
	read (A)
	lock-s (B)
lock-x (A)	

- Neither T_3 nor T_4 can make progress: executing **lock-S**(B) causes T_4 to wait for T_3 to release its lock on B, while executing **lock-X**(A) causes T_3 to wait for T_4 to release its lock on A.
- Such a situation is called a **deadlock**. To resolve the deadlock, one of T_3 or T_4 must be rolled back and its locks released.

Pitfalls of Lock-Based Protocols (2)

- Some locking protocols also permit **starvation**, which means that a transaction does not make progress because it cannot obtain a particular lock. Example:
  - A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
- In practical terms, starvation means that a transaction appears to hang, similarly to deadlock.
- An ideal lock manager prevents both deadlock and starvation.

The Two-Phase Locking Protocol

- **Two-phase locking (2PL):** a popular pessimistic protocol that ensures conflict-serializable schedules.
- High-level idea:
  - **Phase 1:** Growing Phase
    - transaction may acquire locks
    - transaction may not release locks
  - **Phase 2:** Shrinking Phase
    - transaction may release locks
    - transaction may not acquire locks
- The sequence of instructions executed by a transaction determines which items must be locked and in what mode (*e.g.*, request X-lock before writing an item).

2PL Details

- To access a data object, a transaction must hold a sufficiently strong lock on that object. An exclusive lock must be held for writing, but a shared lock is sufficient for reading.
- The protocol allows **lock conversions**. That is, during the growing phase a transaction may **upgrade** a shared lock to an exclusive lock without releasing the lock. Similarly, during the shrinking phase a transaction may **downgrade** an exclusive lock to a shared lock without releasing the lock.
- “Two-phase” means that after a transaction releases or downgrades any lock, it cannot acquire or upgrade the same lock or any other lock. In particular, all **lock-s**, **lock-x**, and **upgrade** instructions must precede any **unlock** and **downgrade** instructions within the same transaction.
- Aside from the above, the protocol does not specify exactly when locks are acquired or in what mode. In principle 2PL allows locks to be acquired long before they are needed, and released long after they are no longer needed.

Properties of 2PL

- **Note 1:** Deadlock is possible under two-phase locking.
- **Note 2:** Cascading rollback is possible under two-phase locking.
- To avoid cascading rollback, one can follow a modified protocol called **strict two-phase locking (S2PL)**:
  - A transaction *holds* all its *exclusive* locks until it commits or aborts.
  - This ensures that schedules are both cascadeless and recoverable.
- Many database systems actually use a slightly stronger protocol called **rigorous two-phase locking (R2PL)**:
  - A transaction *holds* all its locks, *shared or exclusive,* until it commits or aborts.
  - Compared to S2PL, this reduces parallelism slightly but simplifies the implementation.

Automatic Acquisition of Locks

- Using **automatic acquisition of locks**, a transaction T_i issues the usual read/write instructions without explicit locking calls.
- The instruction **read**(D) is processed as:

```
if T_i has a lock on D then
        read(D)
else
        if necessary wait until no other transaction has an X-lock on D
        grant T_i an S-lock on D
        read(D)
end if
```

Automatic Acquisition of Locks (Cont.)

- The instruction **write**(*D*) is processed as:

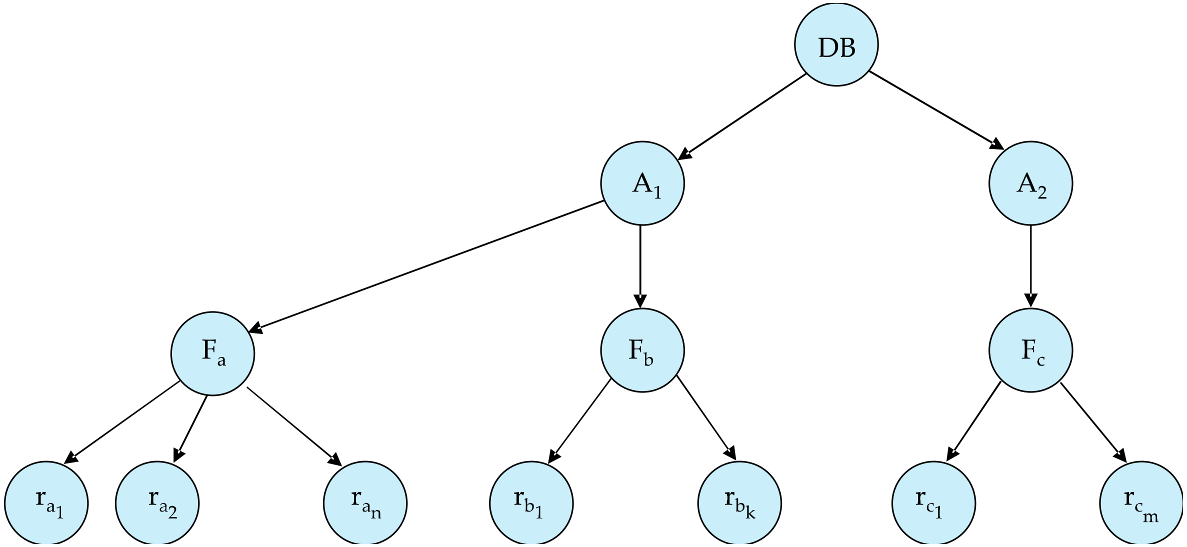
```
if $T_i$ has a X-lock on D then
    write(D)
else
    if necessary wait until no other transaction has any lock on D
    if $T_i$ has an S-lock on D then
        upgrade lock on D  to X-lock
    else
        grant $T_i$ an X-lock on D
    end if
    write(D)
end if
```

- **Note:** All locks are released after commit or abort (for rigorous 2PL).

Multiple Granularity Locking

- Allow data items to be of various sizes and define a hierarchy of data granularities. (Example on next slide.)
- When a transaction locks a node in the tree *explicitly*, it *implicitly* locks all the node's descendants in the *same* mode.
- **Granularity of locking:** the level in the hierarchy, represented as a tree, where locking is done.
  - **Fine granularity** (lower in the hierarchy): greater concurrency, higher locking overhead.
    - Example: locking an individual record.
  - **Coarse granularity** (higher in the hierarchy): less concurrency, lower locking overhead.
    - Example: locking the entire database.

Example of Granularity Hierarchy



In this textbook example, the levels of the tree from top to bottom are:

- *database* (coarsest, global lock)
- *area* (*e.g.*, representing a set of tables belonging to one application)
  - *tablespace/data partition*
- *file* (*e.g.*, representing one table)
  - *table*
- *record* (finest)

  In major databases other levels include

  - *page/block*
  - *index locks (e.g., locking a hash bucket when it is being split)*

Intention Lock Modes

- In addition to S and X lock modes, there are three additional lock modes to support multiple granularity:
  - ***intention-shared*** (IS): indicates intent to lock explicitly at a lower level of the tree but only with shared locks.
  - ***intention-exclusive*** (IX): indicates intent to lock explicitly at a lower level of the tree with exclusive or shared locks.
  - ***shared and intention-exclusive*** (SIX): locks the subtree rooted at a given node explicitly in shared mode and indicates intent to lock explicitly at a lower level with exclusive locks.
- Intention locks allow a higher level node to be locked in S or X mode without having to check all descendant nodes.

Compatibility Matrix with Intention Lock Modes

- Compatibility matrix for all lock modes:

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

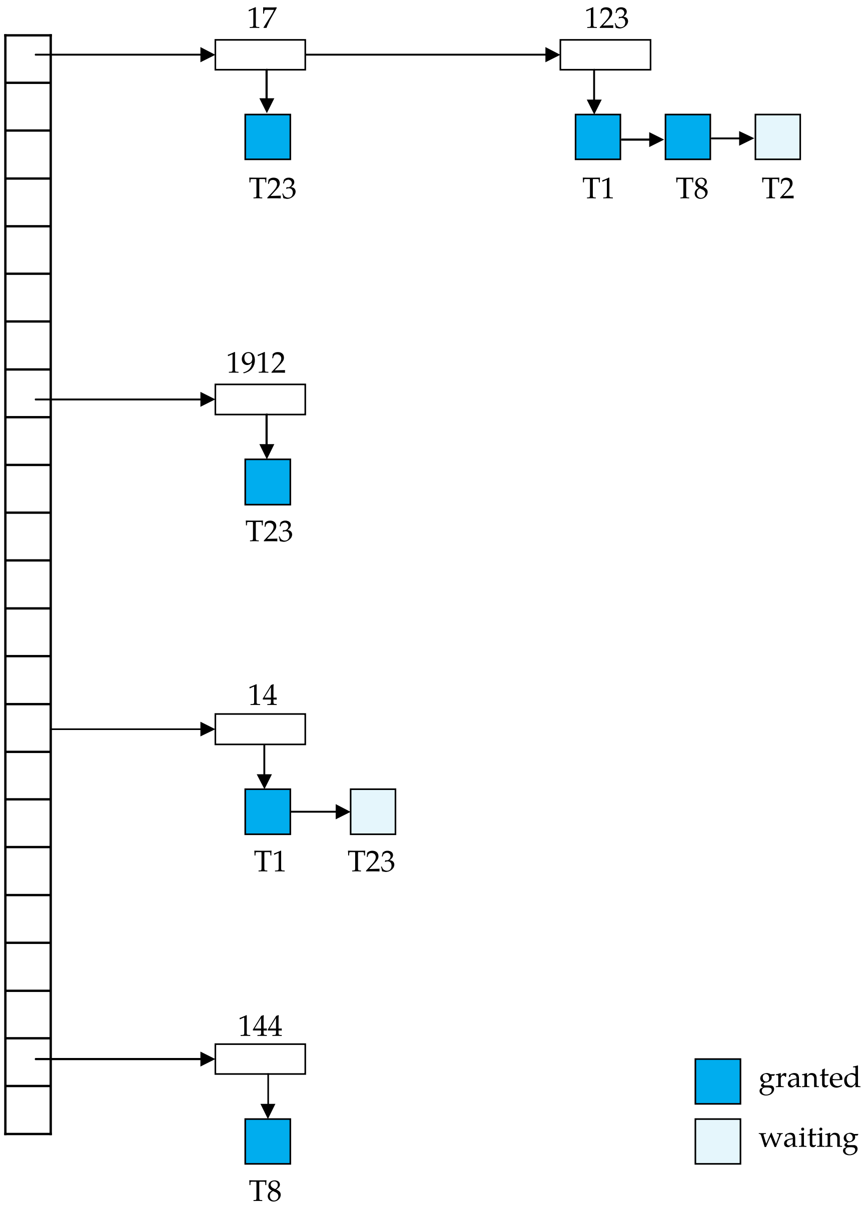
Multiple Granularity: Protocol Details

- Transaction T_i can lock a node Q in the tree using the following rules:
  1. The lock compatibility matrix must be observed.
  2. The root of the tree must be locked first, and may be locked in any mode.
  3. A node Q can be locked by transaction T_i in S or IS mode only if the parent of Q is currently locked by T_i in either IX or IS mode.
  4. A node Q can be locked by transaction T_i in X, SIX, or IX mode only if the parent of Q is currently locked by T_i in either IX or SIX mode.
  5. Transaction T_i can lock a node only if it has not previously unlocked any node (same as in 2PL).
  6. Transaction T_i can unlock a node Q only if none of the children of Q are currently locked by T_i.
- **Note:** locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order.
- **Lock granularity escalation:** in case there are too many locks at a particular level, it is possible to switch to a higher granularity S or X lock.

Implementation of Locking

- Transactions send lock and unlock requests to a **lock manager**.
- The lock manager replies to a lock request with a lock grant message, or a message asking the transaction to roll back instead (to avoid deadlock).
- The requesting transaction waits until its request is answered.
- The lock manager maintains a data-structure called a **lock table** to record granted locks and pending requests
- The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked.

Lock Table



- Darker rectangles indicate granted locks, lighter ones indicate waiting requests
- Lock table also records the type of lock granted or requested
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks.
- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted.
- If a transaction aborts, all waiting or granted requests of the transaction are deleted. For this purpose the lock manager may keep a list of locks held by each transaction.

Multiversion Concurrency Control

- Multiversion concurrency control (MVCC) schemes retain old versions of data items. This allows transactions to access different versions in parallel, increasing concurrency.
- Each successful **write** results in the creation of a new version of the data item written. Versions are labeled using timestamps.
- When a **read**(*Q*) operation is issued, select an appropriate version of *Q* based on the timestamp of the transaction, and return the value of the selected version.
- **Observation: read** operations never have to wait as an appropriate version is returned immediately.

Multiversion Two-Phase Locking

- Differentiates between *read-only transactions* and *update transactions*
- **Update transactions** acquire read and write locks, and hold all locks up to the end of the transaction. That is, update transactions follow rigorous two-phase locking.
  - Each successful **write** results in the creation of a new version of the data item written.
  - Each version of a data item has a single timestamp whose value is obtained from a timestamp counter (called **ts-counter**) that is incremented during commit processing.
- **Read-only transactions** obtain a timestamp when they become active by reading **ts-counter**. When a read-only transaction wants to **read** a data item, it returns the version with the highest timestamp that does not exceed the transaction's timestamp.

Multiversion Two-Phase Locking (2)

- When an update transaction wants to **read** a data item:
  - it obtains an S-lock on the data item, and reads the latest version
- When an update transaction wants to **write** an item:
  - it first obtains an X-lock on the data item
  - it then creates a new version of the item and sets this version's timestamp to ∞.
- When update transaction T_i completes, commit processing occurs:
  - T_i sets timestamp on the versions it has created to **ts-counter** + 1
  - T_i increments **ts-counter** by 1
- Read-only transactions that start after T_i increments **ts-counter** will see the values updated by T_i.
- Read-only transactions that start before T_i increments **ts-counter** will see values that pre-date the updates applied by T_i.
- The protocol ensures conflict serializability, just like ordinary 2PL.

MVCC: Implementation Issues

- Creation of multiple versions increases storage overhead:
  - additional tuples must be stored
  - additional space is required in each tuple for version information
- However, old versions can be garbage collected.
  - *E.g.*, if Q has two versions Q5 and Q9, and the oldest active transaction has timestamp > 9, then Q5 is no longer needed

Demo of MVCC in MySQL / InnoDB

Transaction 1:

- set session transaction isolation level serializable;
- set autocommit = 0;
- start transaction;
- update student set tot_cred = 120 where ID = 00128;
  (holds exclusive lock on ID = 00128)

Transaction 2:

- set session transaction isolation level serializable;
- set autocommit = 1;
- select * from student where ID = 00128;
  (read-only transaction, proceeds without waiting for lock on ID = 00128)

Timestamp Ordering Protocol

- **It is possible to achieve serializability without using locks!**
- High-level idea: Instead of using locks to prevent conflicting accesses to data, transactions use timestamps to detect such conflicts as they are about to occur. The transaction that discovers the conflict resolves it by aborting itself. This can lead to **starvation**.
- Each transaction is issued a timestamp when it enters the system. If an old transaction T_i has timestamp TS(T_i), a newer transaction T_j is assigned a timestamp TS(T_j) > TS(T_i).
- Furthermore, two timestamps are recorded for each data item *Q*:
  - **W-timestamp**(*Q*) is the largest timestamp of any transaction that executed **write**(*Q*) successfully.
  - **R-timestamp**(*Q*) is the largest timestamp of any transaction that executed **read**(*Q*) successfully.

  Note: in this context "successfully" means a transaction issues a read or write, and completes it without detecting a conflict and aborting.

Timestamp Ordering Protocol (Cont.)

- The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order (or not at all).
- Suppose a transaction T_i issues a **read**(Q).

1. If TS(T_i) $<$ W-timestamp(Q), then T_i is attempting to read a value of Q that has already been overwritten.

   In this case the **read** operation is rejected and T_i is rolled back.

2. If TS(T_i) $\geq$ W-timestamp(Q), then T_i is attempting to read the latest value of Q. The read operation returns the value of Q and R-timestamp(Q) is set to **max**(R-timestamp(Q), TS(T_i)).

   In this case the **read** operation is executed successfully.

Timestamp Ordering Protocol (Cont.)

- Suppose that transaction T_i issues **write**(Q).

1. If TS(T_i) < R-timestamp(Q), then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced.

   In this case the **write** operation is rejected and T_i is rolled back.

2. If TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of Q.

   In this case the **write** operation is rejected and T_i is rolled back.

3. Otherwise, the **write** operation updates Q and W-timestamp(Q) is set to TS(T_i).

   In this case the **write** operation is executed successfully.

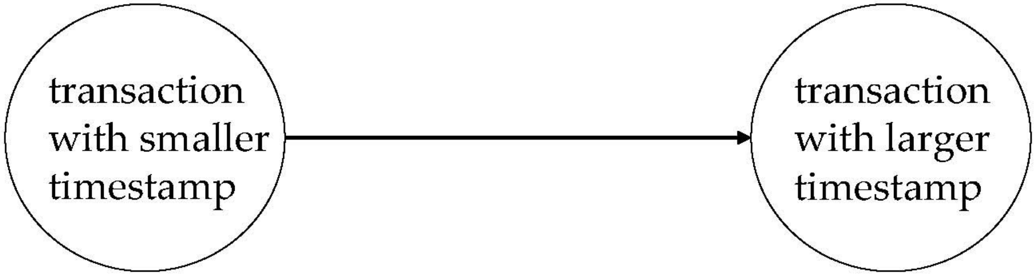
Example of Timestamp Ordering

A partial schedule for transactions T_1, T_2, T_3, T_4, and T_5 with timestamps 1, 2, 3, 4, 5, respectively.

T_1	T_2	T_3	T_4	T_5
				read (X)
	read (Y)			
read (Y)				
		write (Y)		
		write (Z)		
				read (Z)
	read (Z)			
	abort			
read (X)				
			read (W)	
		write (W)		
		abort		
				write (Y)
				write (Z)

Correctness of Timestamp Ordering Protocol

- The timestamp ordering protocol, as presented up to this point, guarantees the edges in the precedence graph are of the form:


transaction with smaller timestamp
transaction with larger timestamp

- Since edges always point from older transactions to newer ones, there can be no cycles in the precedence graph. **Thus, timestamp ordering guarantees conflict serializability.**
- Timestamp ordering also ensures freedom from deadlock since no transaction ever waits for another (*e.g.*, to release a lock).
- But the schedule may not be cascadeless, and may not even be recoverable. Furthermore, starvation may occur if the same transaction (*e.g.*, large read-only transaction) aborts repeatedly.

Recoverability and Cascade-Freedom

- Problem with timestamp ordering protocol:
  - Suppose T_i aborts, but T_j has read a data item written by T_i.
  - Then T_j must abort; if T_j had been allowed to commit earlier, the schedule is not recoverable.
  - Furthermore, any transaction that has read a data item written by T_j must abort. This can lead to cascading rollback.
- **Solution 1:** A transaction is structured such that all of its writes are performed at the end, atomically with the commit step.
  Ensures cascadeless schedules.
- **Solution 2:** Keep track of commit dependencies and delay commitment of a transaction until all of the transactions it depends on have committed. Ensures recoverable schedules.

Thomas' Write Rule

- Modified version of the timestamp ordering protocol in which obsolete **write** operations may be ignored under certain circumstances.
- When T_i attempts to write data item Q, if TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of Q. Rather than rolling back T_i, we can simply ignore the **write** operation (*i.e.*, treat it as a no-op) provided that TS(T_i) ≥ R-timestamp(Q).
- This modification to the timestamp ordering protocol is called **Thomas' write rule**. It enables slightly greater concurrency, and **permits some view-serializable schedules that are not conflict-serializable, such as the following:**

T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

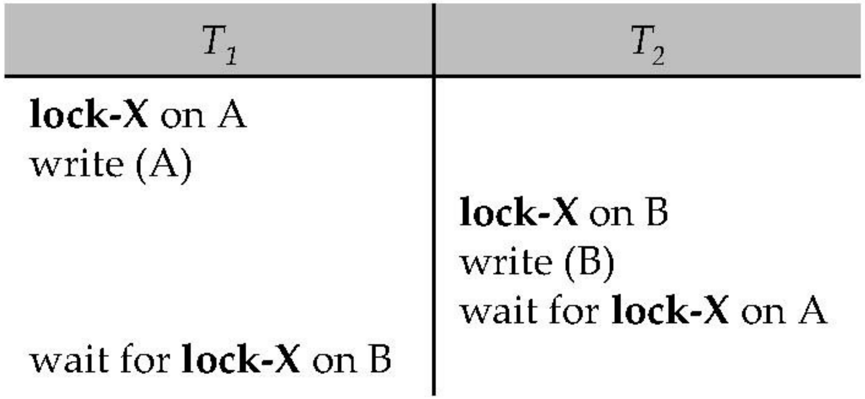
Deadlock Handling

- Consider the following two transactions:

  T_1: write(X) T_2: write(Y)

  write(Y) write(X)

- Hypothetical schedule with deadlock:



T_1	T_2
lock-X on A	
write (A)	
	lock-X on B
	write (B)
	wait for **lock-X** on A
wait for **lock-X** on B	

Deadlock in MySQL / InnoDB

Transaction 1:

- set session transaction isolation level serializable;
- set autocommit = 0;
- start transaction;
- select * from student where ID = 00128;
- update student set tot_cred = 121 where ID = 99999;

ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction

Transaction 2:

- set session transaction isolation level serializable;
- set autocommit = 0;
- start transaction;
- select * from student where ID = 99999;
- update student set tot_cred = 120 where ID = 00128;

Deadlock Handling

- System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.
- ***Deadlock prevention*** protocols ensure that the system will *never* enter into a deadlock state.
- Simple deadlock prevention strategies include the following:
  - Pre-declaration: require that each transaction locks all its data items before it begins execution.
  - Lock ordering: impose a partial order on all data items and require that a transaction lock data items in the order specified.
  - Timeout-based: a transaction waits for a lock only for a specified amount of time, and then rolls back if lock is not granted.
- ***Deadlock detection*** protocols allow deadlock to occur, detect when it has occurred, and then recover by rolling back some transaction.

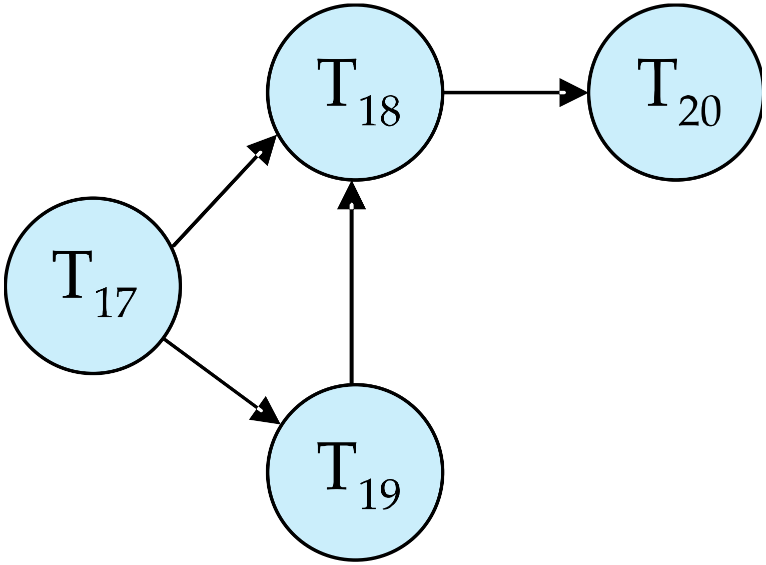
More Deadlock Prevention Strategies

- More elaborate deadlock prevention strategies incorporate transaction timestamps that indicate the time when a transaction became active. (Similar timestamps are used by the timestamp ordering protocol.)
- **Wait-die** scheme (non-preemptive):
  - Older transaction may wait for younger one to release a lock. Younger transactions never wait for older ones; they are rolled back (*i.e.*, die) instead.
  - A transaction may die several times before acquiring a needed lock.
- **Wound-wait** scheme (preemptive):
  - Older transaction *wounds* (forces rollback of) younger transaction instead of waiting for it. Younger transactions may wait for older ones.
  - Potentially fewer rollbacks than in *wait-die* scheme.
  - Used by Google's Spanner database!
- **Note:** both "wait-die" and "wound-wait" avoid starvation.

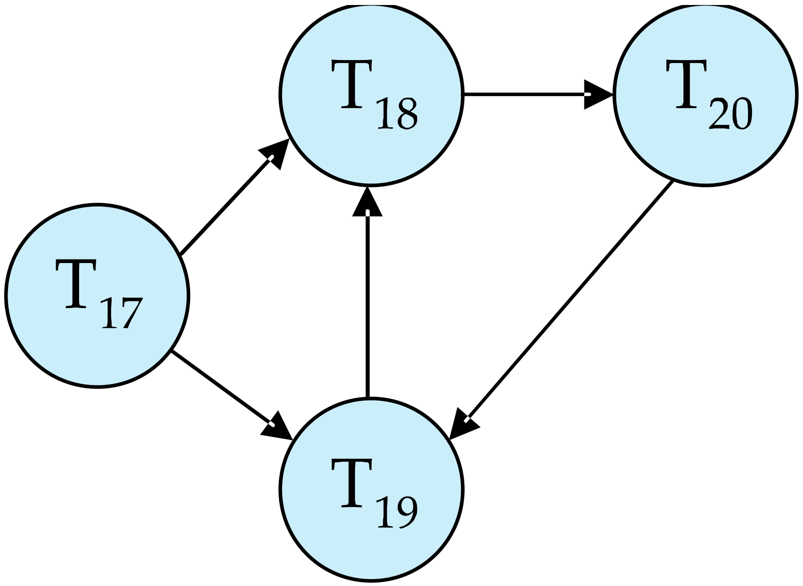
Deadlock Detection

- Deadlocks can be detected by constructing a **wait-for graph** *G*(*V*, *E*) where:
  - *V* is a set of vertices representing the transactions in the system
  - *E* is a set of directed edges representing wait-for dependencies where $T_i \rightarrow T_j$ is in *E* if T_i is waiting for T_j to release a data item
- When T_i requests a data item currently being held by T_j, then the edge $T_i \rightarrow T_j$ is inserted into the wait-for graph. This edge is removed only when T_i stops waiting, such as when T_j is no longer holding a lock on a data item needed by T_i.
- **Observation:** The system is in a deadlock state if and only if the wait-for graph has a cycle.
- The DBMS (or a particular storage engine) must invoke a deadlock detection algorithm periodically to check for cycles.

Deadlock Detection (Cont.)



Wait-for graph without a cycle.
(no deadlock)



Wait-for graph with a cycle.
(deadlock)

Deadlock Recovery

- When deadlock is detected:
  - Some transaction must be rolled back (*i.e.*, made a victim) to break the deadlock.
  - The victim is chosen to be the transaction whose rollback will incur minimum cost.
  - Starvation happens if the same transaction is chosen as victim repeatedly. The number of rollbacks can be included in the cost calculation to avoid starvation.
  - The victim may be rolled back either partially or completely.
    - **Total rollback:** abort the transaction and then restart it.
    - **Partial rollback:** roll back the transaction only as far as necessary to break the deadlock.

Database Recovery

- **Transaction failure:**
  - **Logical errors:** transaction cannot complete due to some internal error condition (*e.g.*, division by zero, or attempt to violate referential integrity).
  - **System errors:** the database system must terminate an active transaction due to an error condition (*e.g.*, deadlock).
- **System crash:** a power failure or other hardware or software failure causes the entire system to crash, and possibly reboot later.
  - **Fail-stop assumption**: non-volatile storage contents are not corrupted by a system crash.
- **Disk failure:** a head crash or other hardware failure destroys all or part of a disk.
  - Destruction is assumed to be detectable: disk drives use checksums to detect failures.

Technical Challenges in Recovery

- Consider a transaction T_i that transfers \$50 from account *A* to account *B*. This transaction must execute two updates: subtract 50 from *A* and then add 50 to *B*. When the transaction commits, these updates must persist in non-volatile storage for durability (D in ACID).
- Achieving atomicity and durability <u>simultaneously</u> is difficult because modifications to *A* and *B* may correspond to separate I/O operations.
  - A failure may occur after one of the modifications has been made but before both of them are made.
  - Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state.
  - Not modifying the database may result in lost updates if failure occurs just after transaction commits.
- Recovery algorithms have two parts:
  1. Actions taken during normal transaction processing to ensure that sufficient housekeeping information exists to recover from failures.
  2. Actions taken after a failure to recover the database contents to a state that ensures ACID properties.

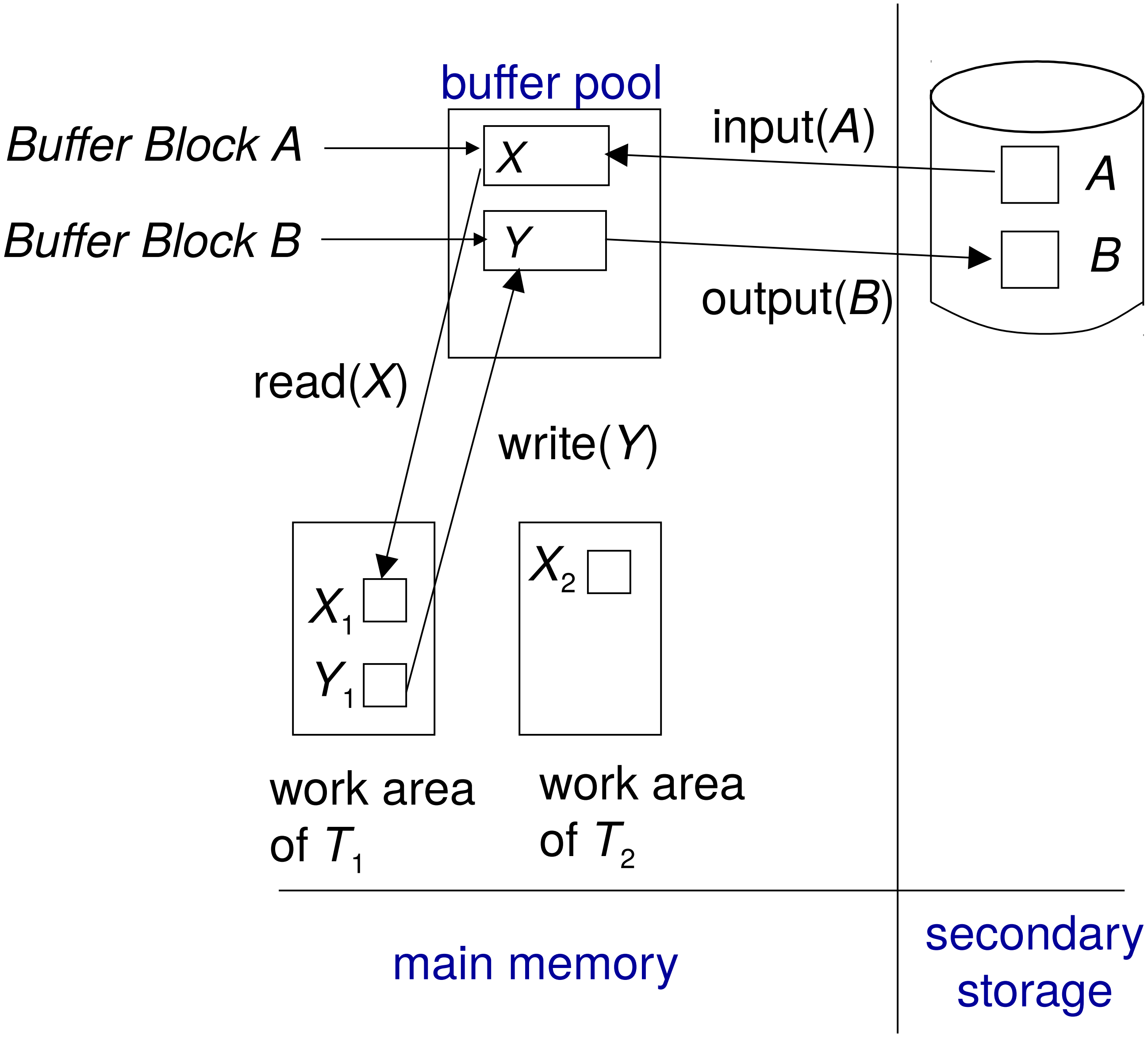
Storage Structure

- **Volatile storage**:
    - does not survive system crashes
    - examples: main memory, cache memory
- **Non-volatile storage**:
    - survives system crashes
    - examples: hard disk, solid state drive, battery-backed RAM
    - but may still fail, losing data (*e.g.*, head crash, dead battery)
- **Stable storage**:
    - a hypothetical form of storage that survives all failures
    - approximated in reality by maintaining multiple copies of data on distinct nonvolatile media (*e.g.*, using RAID)
    - plays a crucial role in the database recovery!

Data Access

- **Physical blocks** are those blocks residing on the disk.
- **Buffer blocks** are the blocks residing temporarily in main memory (*i.e.*, buffered in the buffer pool).
- The buffer pool manager moves blocks between disk and main memory using the following two operations:
  - **output**(*B*) transfers the buffer block *B* from main memory to disk, overwriting the corresponding physical block on disk
  - **input**(*B*) transfers the physical block *B* from disk to main memory (if necessary, free space in the buffer pool can be created by first evicting an existing buffer block and writing it to disk)
- We assume, for simplicity, that each data item is stored inside a single block. A real DBMS must accommodate large data items that span multiple blocks (*e.g.*, see “BLOB” data type in SQL).

Example of Data Access



Data Access (Cont.)

- Each transaction T_i has a private **work area** in which it keeps local copies of all data items it accesses and updates.
  - T_i's local copy of a data item X is called X_i.
- Transferring data items between buffer blocks and the private work area done using two operations:
  - **read**(X) assigns the value of data item X to the local variable X_i
  - **write**(X) assigns the value of local variable X_i to data item X in the buffer block
  - **Note:** In the previous slide **output**(B) need not immediately follow **write**(X). The buffer pool manager can delay the **output** operation to improve I/O performance.
- Transaction behavior:
  - must perform **read**(X) before accessing X for the first time (subsequent reads can be from local copy in work area)
  - may execute **write**(X) at any time before the transaction commits

Recovery and Atomicity

- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.
- In this lecture module we will focus exclusively on **log-based recovery** mechanisms.
  - We will cover only fundamental concepts:
    - **write-ahead logging (WAL)**
    - **redo and undo operations**
  - Real-world databases use more elaborate log-based recovery algorithms such as **ARIES** (Algorithms for Recovery and Isolation Exploiting Semantics) and incorporate a number of important optimizations.
- Less frequently used alternative: **shadow-paging** (see textbook).

Log-Based Recovery

- A **recovery log** is a sequential (*i.e.*, “append-only”) structure recorded in stable storage. It comprises a sequence of **log records**, and represents a history of updates to the database.
- When transaction T_i starts, a log record $\langle T_i$ **start**$\rangle$ is written.
- <u>Before</u> T_i executes **write**(X), a log record of the form

  $$\langle T_i, X, V_1, V_2 \rangle$$

  is written, where V_1 is the value of X before the write (the **old value**), and V_2 is the value to be written to X (the **new value**).
- When T_i finishes it last statement, the log record

  $\langle T_i$ **commit**$\rangle$

  is written.
- A transaction is said to have committed when its commit log record (and all prior log records) is output to stable storage.

Immediate *vs.* Deferred Modification

- Although all updates persist in the recovery log, we still need to update the state of the database on disk, for example by modifying table rows and index entries. This allows us to answer queries efficiently, without having to replay the recovery log.
- The **immediate-modification** scheme allows updates of an uncommitted transaction to be made to the buffer, or the disk itself, before the transaction commits.
  - **Write-ahead logging (WAL) rule:** Update log record must be output to stable storage *before* database item is written to disk.
  - Output of updated blocks to disk can take place at any time before or after transaction commit. (Not always immediately!)
  - Order in which blocks are output (to disk) can be different from the order in which they are written to (buffer pool).
- The **deferred-modification** scheme writes to buffer/disk only at the time of transaction commit.
  - This simplifies some aspects of recovery, but imposes the overhead of storing a local copy.

Immediate Database Modification Example

Log	**Write**	**Output**
<T_0 **start**>		
<T_0, X, 1000, 950>		
	X = 950	
<T_0, Y, 2000, 2050>		
	Y = 2050	
<T_0 **commit**>		
<T_1 **start**>		
<T_1, Z, 700, 600>		
	Z = 600	
		B_Y , B_Z
<T_1 **commit**>		
		B_X

Note: B_X denotes block containing X.

B_Z output before T_1 commits

B_X output after T_0 commits

Immediate *vs.* Deferred Modification

- Although all updates persist in the recovery log, we still need to update the state of the database on disk, for example by modifying table rows and index entries. This allows us to answer queries efficiently, without having to replay the recovery log.
- The **immediate-modification** scheme allows updates of an uncommitted transaction to be made to the buffer, or the disk itself, before the transaction commits.
  - **Write-ahead logging (WAL) rule:** Update log record must be output to stable storage *before* database item is written to disk.
  - Output of updated blocks to disk can take place at any time before or after transaction commit. (Not always immediately!)
  - Order in which blocks are output (to disk) can be different from the order in which they are written to (buffer pool).
- The **deferred-modification** scheme writes to buffer/disk only at the time of transaction commit.
  - This simplifies some aspects of recovery, but imposes the overhead of storing a local copy.

Undo and Redo Operations

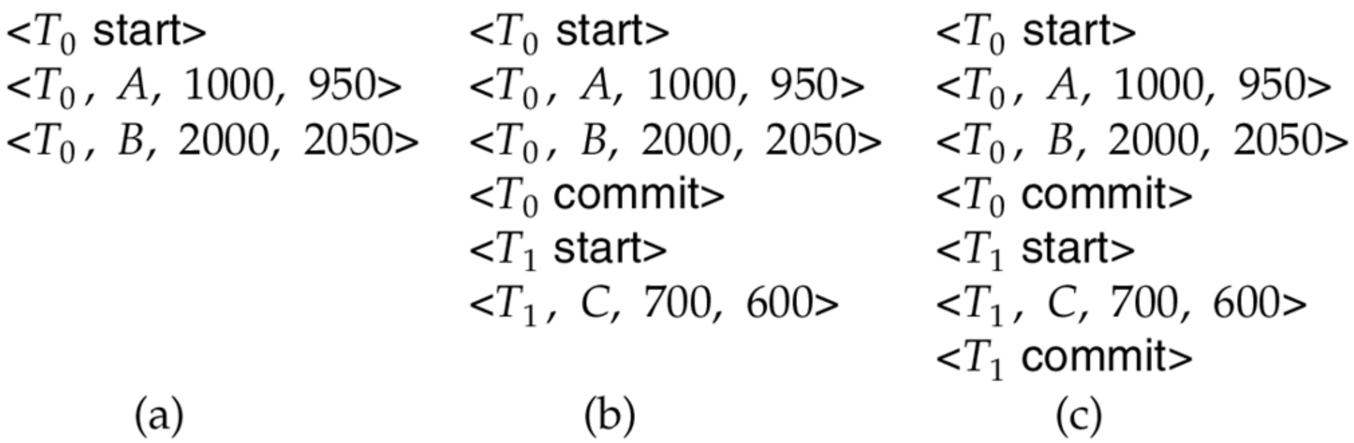
- On recovery, any transactions that did not finish prior to the failure (*i.e.*, did not commit or abort) must be cleaned up. This involves undoing/redoing entire transactions as well as individual log records.
- **For log records:**
  - **Undo** of <T_i, X, V_1, V_2> executes **write**(X) with value V_1.
  - **Redo** of <T_i, X, V_1, V_2> executes **write**(X) with value V_2.
- **For transactions:**
  - **Undo** of T_i undoes all the log records corresponding to updates applied by T_i by going backward from the last log record of T_i.
    - Each time a log record <T_i, X, V_1, V_2> is undone, a special **compensation log record** <T_i, X, V_1> is appended to the log.
    - The procedure ends by appending <T_i **abort**> to the log.
  - **Redo** of T_i redoes all the log records corresponding to updates applied by T_i (including any redo log records) by going forward from the first log record for T_i.
    - There is no additional logging in this case!

Undo and Redo on Recovering from Failure

- In our simplified discussion of recovery all transactions in the recovery log are either redone or undone after a failure. A better algorithm is described later on that only considers a subset of the transactions in the log.
- Transactions are dealt with as follows on recovery:
  - Transaction T_i needs to be **undone** if the log
    - contains the record <T_i **start**>,
    - but does not contain either the record <T_i **commit**> *or* <T_i **abort**>.
  - Transaction T_i needs to be **redone** if the log
    - contains the record <T_i **start**>
    - and contains the record <T_i **commit**> *or* <T_i **abort**>
  - **Note:** If T_i is undone then it will end up in the aborted state. Otherwise if T_i is redone then it may end up in the committed *or* aborted state.
- Note that if T_i was undone and the <T_i **abort**> record written to the log prior to the failure, then on recovery T_i may be redone anyway including all the original actions *as well as any compensation log records.* This technique, known as **repeating history,** simplifies the implementation of recovery.

Immediate DB Modification Recovery Example

Below we show the log as it appears at three instances of time.



<T_0 start>
<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>

(a)

<T_0 start>
<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>
<T_0 commit>
<T_1 start>
<T_1, C, 700, 600>

(b)

<T_0 start>
<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>
<T_0 commit>
<T_1 start>
<T_1, C, 700, 600>
<T_1 commit>

(c)

Recovery actions in each case above are:

(a) Undo (T_0): *B* is restored to 2000 and *A* to 1000, and log records <T_0, *B*, 2000>, <T_0, *A*, 1000>, <T_0, **abort**> are written out.

(b) Redo (T_0) and undo (T_1): *A* and *B* are set to 950 and 2050 and *C* is restored to 700. Log records <T_1, C, 700>, <T_1, **abort**> are written out.

(c) Redo (T_0) and redo (T_1): *A* and *B* are set to 950 and 2050 respectively, then *C* is set to 600.

Checkpoints

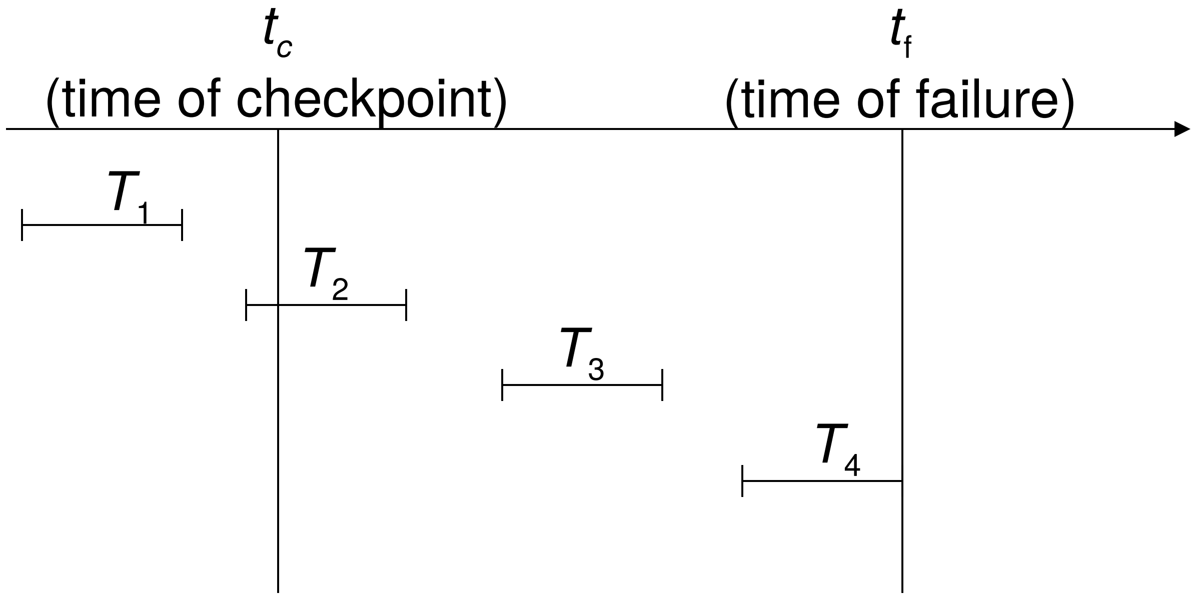
- Redoing/undoing *all* transactions recorded in the recovery log can be painfully slow.
    1. Processing the entire log is time-consuming if the system has run for a long time and the log is very long (*e.g.*, Gigabytes).
    2. We might unnecessarily redo transactions that have already finished *and* output their updates to the database.
- Real world databases streamline the recovery procedure by periodically saving a **checkpoint**:
    1. If any log records are buffered in main memory then output them to stable storage. (See later discussion of log record buffering.)
    2. Then output any remaining modified buffer blocks to the disk.
    3. Finally write a log record <**checkpoint** *L*> to stable storage where *L* is a list of all transactions active at the time of checkpoint.

    Note: All updates must be stopped while creating the checkpoint. This is very disruptive!

Checkpoints (Cont.)

- During recovery, we process the log roughly from the most recent checkpoint.
  - Scan backward from end of log to find the most recent <**checkpoint** *L*> record.
  - Only transactions that are in *L* or started after the checkpoint need to be redone or undone.
  - Transactions that committed or aborted before the checkpoint already have all their updates output to disk.
- Some earlier part of the log may also be needed for undo operations.
  - During the undo phase (described in more detail later on), continue scanning backwards until a record <T_i **start**> is found for every transaction T_i in *L*.
  - Parts of the recovery log prior to earliest <T_i **start**> record above are not needed for recovery, and can be discarded.

Example of Checkpoints



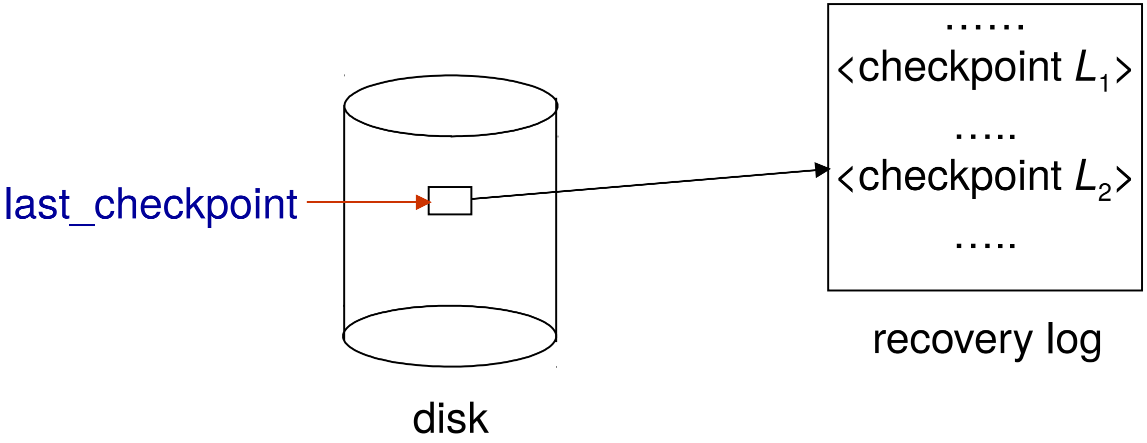
- On recovery T_1 can be ignored.
  (Updates already output to disk due to checkpoint.)
- On recovery T_2 and T_3 are redone.
  (*i.e.*, the steps performed after the checkpoint are replayed.)
- On recovery T_4 is undone.
  (For simplicity we also replay the steps of T_4 before undoing it.)

Fuzzy Checkpointing

- To avoid long interruption of normal processing during checkpointing, allow updates to occur during checkpointing.
- **Fuzzy checkpointing** is done as follows:
  1. Temporarily stop all updates by transactions.
  2. Write a <**checkpoint** *L*> log record and force log to stable storage.
  3. Note list *M* of modified buffer blocks.
  4. Now resume ordinary transaction execution.
  5. Output to disk all modified buffer blocks in list *M.*
     - Blocks should not be updated while being output.
     - Follow WAL rule: all log records pertaining to a block must be output before the block is output.
  6. Store a pointer to the **checkpoint** record in a fixed position **last_checkpoint** on disk.

Fuzzy Checkpointing (Cont.)

- When recovering using a fuzzy checkpoint, start scan from the **checkpoint** record pointed to by **last_checkpoint**.
  - Log records before **last_checkpoint** have their updates reflected in database on disk, and need not be redone.
  - Incomplete checkpoints, where system had crashed while performing checkpoint, do not break ACID properties.



last_checkpoint

……

<checkpoint L_1>

…..

<checkpoint L_2>

…..

recovery log

disk

Log-Record Buffering

- **Log-record buffering**: log records are buffered in main memory, instead of being output directly to stable storage.
- Log records are output to stable storage when a block of log records in the buffer is full, or a **log force** operation is executed.
- Log force is performed to commit a transaction by forcing all its log records (including the commit record) to stable storage.
- Several log records can thus be output using a single output operation, reducing the I/O cost.

Log-Record Buffering (2)

- The rules below must be followed if log records are buffered:
  - Log records are output to stable storage in the order in which they are created.
  - Transaction T_i enters the commit state only when the log record $\langle T_i$ **commit**$\rangle$ has been output to stable storage.
  - We must follow a more general form of the **write-ahead logging** or **WAL** rule: before a block of data in main memory is output to the database, all log records pertaining to data in that block must have been output to stable storage.

Failure of Non-volatile Storage

- So far we assumed no loss of non-volatile storage.
- Techniques similar to checkpointing are used to deal with loss of non-volatile storage, which may destroy the database but not the log.
  - Periodically **dump** the entire content of the database to stable storage. No transaction may be active during the dump procedure. A procedure similar to (non-fuzzy) checkpointing must take place:
    1. Output all log records currently residing in main memory to stable storage.
    2. Output all modified buffer blocks to disk.
    3. Copy the contents of the entire database to stable storage.
    4. Output a record <**dump**> to recovery log on stable storage.
  - Technique can be extended to allow transactions to be active during dump (**fuzzy dump** or **online dump**). Similar to fuzzy checkpointing.
- On recovery, the database is restored from the dump in stable storage, and transactions are recovered using the log as usual.

Simplified Recovery Algorithm: Complete Details and Example

Recovery Algorithm

- **Logging** (during normal operation):
  - append $<T_i$ **start**$>$ at transaction start
  - append $<T_i, X_j, V_1, V_2>$ for each update (from V_1 to V_2), and
  - append $<T_i$ **commit**$>$ at transaction end
- **Transaction rollback** (during normal operation):
  - let T_i be the transaction to be rolled back
  - scan log backwards from the end, and for each log record of T_i of the form $<T_i, X_j, V_1, V_2>$
    - perform the undo by applying **write**(X_j) with value V_1
    - append a compensation log record $<T_i, X_j, V_1>$
  - Once the record $<T_i$ **start**$>$ is found stop the scan and append the log record $<T_i$ **abort**$>$.

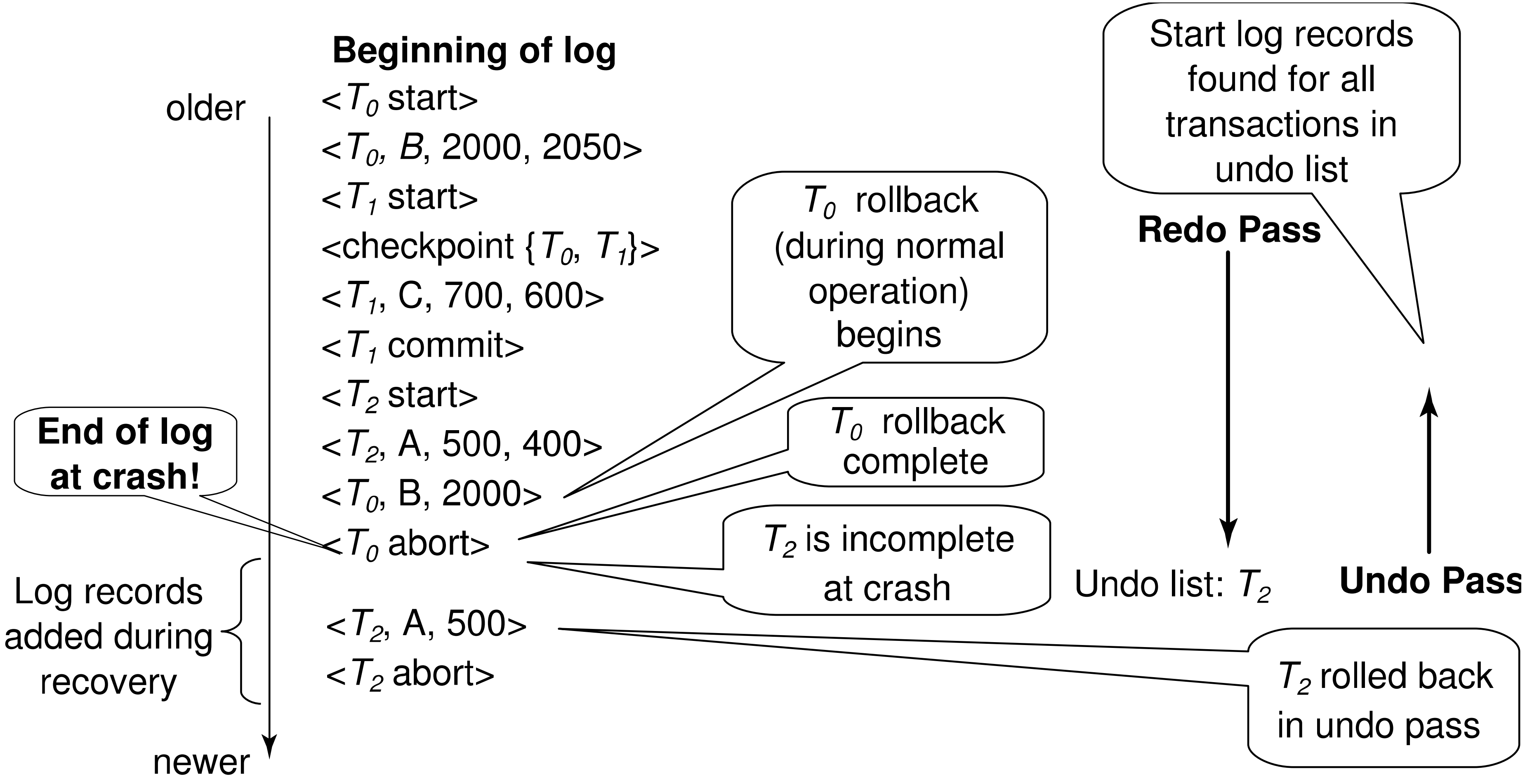
Recovery Algorithm (2)

- **Recovery from failure** (two phases):
  - **Redo phase**: replay updates of *all* transactions since the last checkpoint, regardless of their outcome.
  - **Undo phase**: undo all incomplete transactions.
- **Redo phase**:
  1. Find last <**checkpoint** *L*> record, and set *undo-list* := *L*.
  2. Scan forward from the checkpoint record identified in step 1:
     a) Whenever a record $<T_i, X_j, V_1, V_2>$ is found, redo it by applying **write**(X_j) with value V_2.
     b) Whenever a compensation record $<T_i, X_j, V_1>$ is found, redo it applying **write**(X_j) with value V_1.
     c) Whenever a log record <T_i **start**> is found, add T_i to *undo-list*.
     d) Whenever a log record <T_i **commit**> *or* <T_i **abort**> is found, remove T_i from *undo-list*.

Recovery Algorithm (3)

- **Undo phase:**
  1. scan log backwards from end:
     a) Whenever a log record <T_i, X_j, V_1, V_2> is found where T_i is in *undo-list* perform same actions as for transaction rollback:
        i. perform undo by applying **write**(X_j) with value V_1
        ii. append a log record <T_i , X_j, V_1>
     b) Whenever a log record <T_i **start**> is found where T_i is in *undo-list*:
        i. append a log record <T_i **abort**>
        ii. remove T_i from *undo-list*
     c) Stop when undo-list is empty.
        (*I.e.*, when <T_i **start**> found for every transaction in *undo-list*.)
- Resume ordinary transaction processing after the undo phase ends.

Example of Recovery



Beginning of log

<T_0 start>

<T_0, B, 2000, 2050>

<T_1 start>

<checkpoint {T_0, T_1}>

<T_1, C, 700, 600>

<T_1 commit>

<T_2 start>

<T_2, A, 500, 400>

<T_0, B, 2000>

<T_0 abort>

<T_2, A, 500>

<T_2 abort>

older

newer

End of log at crash!

Log records added during recovery

T_0 rollback (during normal operation) begins

T_0 rollback complete

T_2 is incomplete at crash

T_2 rolled back in undo pass

Start log records found for all transactions in undo list

Redo Pass

Undo list: T_2

Undo Pass