

COMP9311: DATABASE SYSTEMS

Term 1 2024

Week 9 – Transaction Management

By Xiaoyang Wang, CSE UNSW

*Disclaimer: the course materials are sourced from previous offerings of
COMP9311 and COMP3311*

A solid orange horizontal bar at the bottom of the slide.

MyExperience Survey

- The UNSW MyExperience survey is open, participation is highly encouraged.
- *“Please participate in the myExperience Survey and take the opportunity to share your constructive thoughts on your learning experience. Your contributions help your teachers and shape the future of education at UNSW.”*
- You can access the survey by logging into Moodle or accessing myexperience.unsw.edu.au directly.
- More information: <https://www.student.unsw.edu.au/myexperience>

Concurrency Control

Transactions are submitted to the system

How can concurrency control help us produce correct results?

We have categorized schedules that produce correct results

- Serial schedules (not efficient)
- Non-serial schedules
 - (Conflict) Serializable (efficient, correct)
 - Non-(Conflict) serializable (not correct)

Why do we need concurrency control?

Because it is impractical to test serializability on the fly.

Locks

A **lock** is a mechanism to control concurrent access to a data item

Data items can be locked in two modes:

Exclusive (X) mode.

- The data item can be both read as well as written.
- Also known as a **Write Lock**.

Shared (S) mode.

- The data item can only be read.
- Also known as a **Read Lock**.

To use a data item, you must acquire the relevant locks. Transaction can access only after request is granted.

Lock-Based Protocols

Lock requests are made to **concurrency-control manager**.

An exclusive lock is requested using **write_lock()** instruction.

A shared lock is requested using **read_lock()** instruction.

A transaction may be granted a lock on an item if the requested lock is **compatible** with locks already held on the item by other transactions

If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.

Goal: Locking protocols enforce serializability by restricting the set of possible schedules

Lock-Based Protocols

T_1	T_2	concurrency-control manager
<code>write_lock(B)</code> <code>read(B)</code> <code>B := B - 50</code> <code>write(B)</code> <code>unlock(B)</code>	<code>read_lock(A)</code> <code>read(A)</code> <code>unlock(A)</code> <code>read_lock(B)</code> <code>read(B)</code> <code>unlock(B)</code> <code>display(A + B)</code>	<code>grant-write_lock(B, T₁)</code> <code>grant-read_lock(A, T₂)</code> <code>grant-read_lock(B, T₂)</code> <code>grant-write_lock(A, T₁)</code>
<code>write_lock(A)</code> <code>read(A)</code> <code>A := A + 50</code> <code>write(A)</code> <code>unlock(A)</code>		

Note:

- Grants omitted in rest of slides
- Assume grant happens just before the next instruction following lock request

Lock-Based Protocols

Lock-compatibility matrix

	S	X
S	true	false
X	false	false

A transaction may be granted a lock on an item:

- if the requested lock is **compatible** with locks already held on the item by other transactions

This means:

- Any number of transactions can hold shared locks on an item,
- But if a transaction holds an exclusive lock on an item, no other transaction may hold any lock on the item.

Lock-Based Protocols

A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks.

Locking protocol 1: A simple locking protocol

If T has only one operation manipulating an item X:

- if read: obtain a read lock on X before reading
- if write: obtain a write lock on X before writing
- unlock X after this operation on X

Even if T has several operations manipulating X, we obtain **one** lock still:

- if all operations on X are reads, obtain read lock
- if at least **one** operation on X is a write, obtain write lock
- unlock X after the **last** operation on X

Lock-Based Protocols

Simple locking protocol in action:

	T_1	T_2
1	read_lock(Y)	
2	read(Y)	
3	unlock(Y)	
		read_lock(X) 4
		read(X) 5
		unlock(X) 6
		write_lock(Y) 7
		read(Y) 8
		$Y \leftarrow X + Y$ 9
		write(Y) 10
		unlock(Y) 11
12	write_lock(X)	
13	read(X)	
14	$X \leftarrow X + Y$	
15	write(X)	
16	unlock(X)	

Lock-Based Protocols

Example of transactions performing locking:

T1 write_lock(B);
read(B);
B: = B - 50;
write(B);
unlock(B);
write_lock(A);
read(A);
A: = A + 50;
write(A);
unlock(A);

T2 read_lock(A);
read(A);
unlock(A);
read_lock(B);
read(B);
unlock(B);
display(A+B);

Locking as above is not sufficient to guarantee serializability

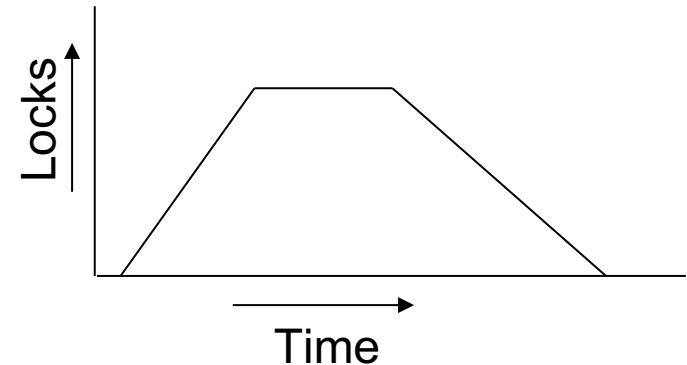
Two Phase Locking (2PL)

Phase 1: Growing Phase

- Transaction obtains locks
- Transaction does not release any locks

Phase 2: Shrinking Phase

- Transaction releases locks
- Transaction does not obtain new locks



This protocol ensures serializability: and produces **conflict-serializable schedules**. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e., the point where a transaction acquired its final lock).

Two Phase Locking (2PL)

Example of transactions performing locking:

T3	write_lock(B);	T4	read_lock(A);
	read(B);		read(A);
	B: = B - 50;		read_lock(B);
	write(B);		read(B);
	write_lock(A);		display(A+B);
	read(A);		unlock(A);
	A: = A + 50;		unlock(B);
	write(A);		
	unlock(B);		
	unlock(A);		

Locking as above is sufficient to guarantee serializability, if unlocking is delayed to the end of the transaction.

Summary

2PL can guarantee serializability, thus allowing real-time control of concurrent executions (scheduling).

Deadlocks

Deadlock occurs when each transaction T in a set of two or more transactions is waiting for some item that is locked by some other transaction T in the set.

In most locking protocols, a deadlock can exist.



Deadlocks

Consider the partial schedule to the right.
The instructions from T3 and T4 arrive at the system...

executing **read_lock(B)**:

- T4 waits for T3 to release its lock on B

executing **write_lock(A)**:

- T3 waits for T4 to release its lock on A

Such a situation is called a **deadlock**.

Issue: neither T3 nor T4 can make progress

	T_3	T_4
1	write_lock(B)	
2	read(B)	
3	$B := B - 50$	
4	write(B)	
5		read_lock(A)
6		read(A)
7		read_lock(B)
8	write_lock(A)	

Deadlock Prevention Scheme

Locking protocols are used in most commercial DBMSs.

We need to address this issue of deadlocks

One way to prevent deadlock is to use a **deadlock prevention protocol**.

Deadlock Prevention Scheme

Timeouts

If a transaction waits for a period longer than a system-defined timeout period, the system assumes that the transaction may be deadlocked and aborts it.

Pro: Small overhead and is simple

Con: There may not be a deadlock

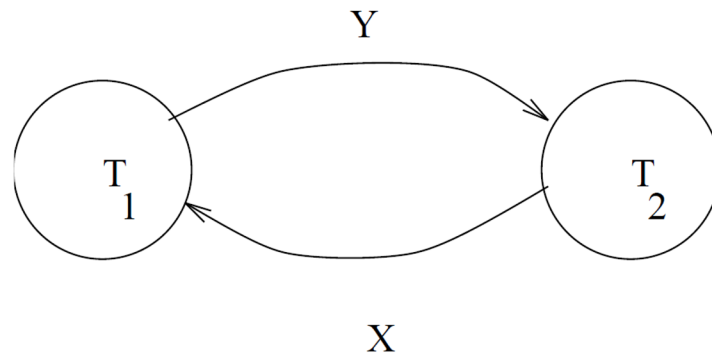
Testing for Deadlocks

Create a **wait-for graph** for currently **active** transactions:

- create a vertex for each transaction; and
- an arc from T_i to T_j if T_i **is waiting** for an item locked by T_j .

If the graph has a cycle, then a deadlock has occurred.

Example:



Testing for Deadlocks

Case: wait-for graph with cycle(s)

	T_3	T_4
1	write_lock(B)	
2	read(B)	
3	$B := B - 50$	
4	write(B)	
5		read_lock(A)
6		read(A)
7		read_lock(B)
8	write_lock(A)	

Summary

- The use of locks, combined with the 2PL protocol, guarantees serializability of schedules.
- The serializable schedules produced by 2PL have their equivalent serial schedules based on the order in which executing transactions lock the items they acquire.
- If a transaction needs an item that is already locked, it may be forced to wait until the item is released. Some transactions may be aborted and restarted because of the deadlock problem.

Transaction Failures

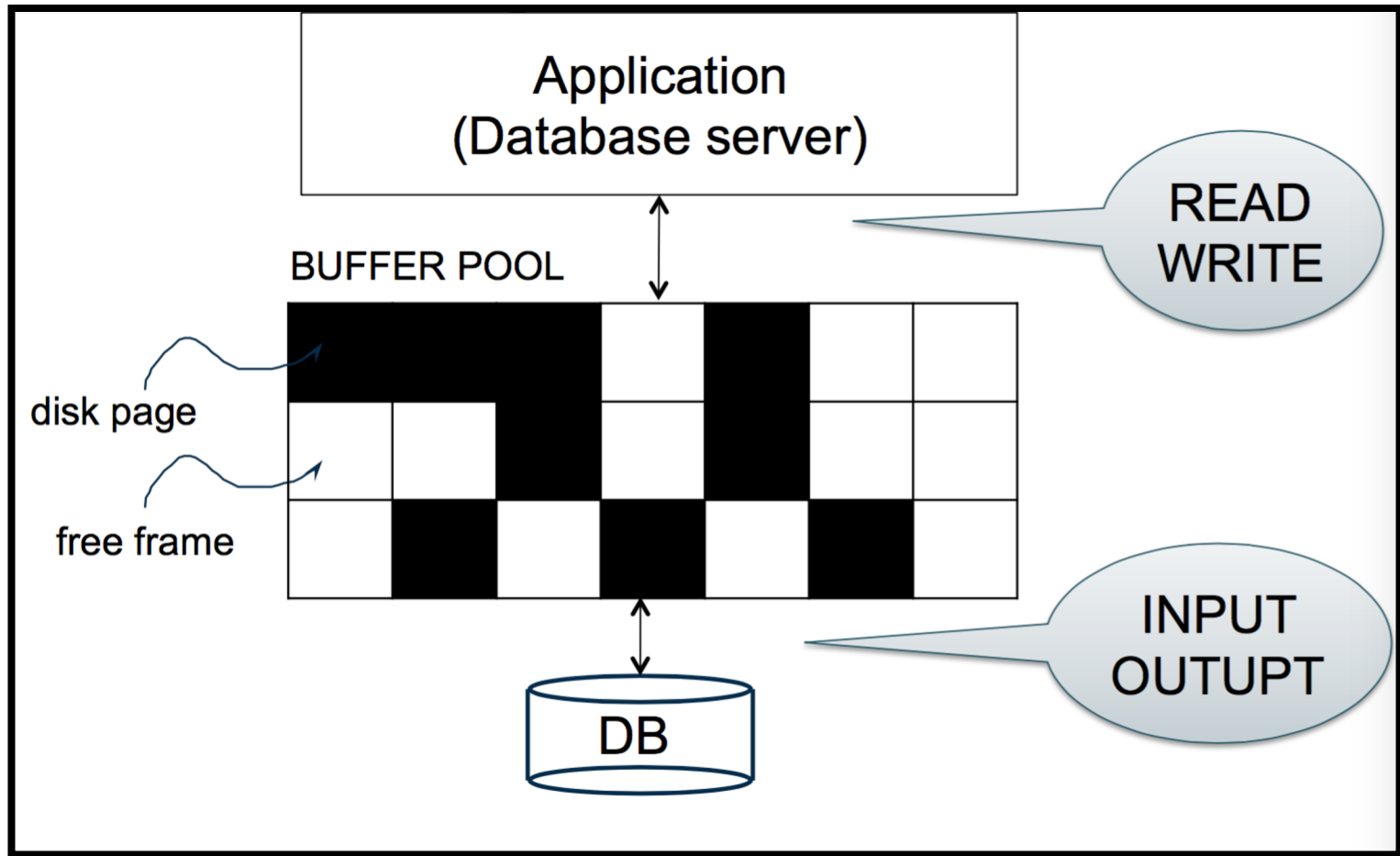
Up to now, we discuss schedules assuming implicitly that there are no **transaction failures**.

- We need to consider the effect of transaction failures during concurrent execution.
- Why? If a transaction T_i fails, we need to **undo** the effect of this transaction to ensure the atomicity property.
- By atomicity, it requires that any transaction T_j that is dependent on T_i (i.e., T_j has read data written by T_i) is also aborted.

Recall Failures

- **A computer failure (system crash).** A hardware, software, or network error occurs in the computer system during transaction execution.
- **A transaction or system error.** Some operation in the transaction may cause it to fail.
- **Local errors or exception conditions detected by the transaction.**
During transaction execution, certain conditions may occur that necessitate cancellation of the transaction
- **Concurrency control enforcement.** The concurrency control method may decide to abort a transaction because it violates serializability or it may abort one or more transactions to resolve a state of deadlock among several transactions.

Buffer Management in a DBMS



Database Recovery

- The recovery strategy is to identify any changes that may cause an inconsistency in the database.
- We need to have information to rollback an unsuccessful transaction (undo any partial updates).

System Log

To be able to recover from failures that affect transactions, the system maintains a **log** to keep track of all transaction operations that affect the values of database items.

- The system needs to record the states information to recover failures correctly.
- The information is maintained in a log (also called journal).
- The system log is kept in hard disk but maintains its current contents in main memory.

System Log

Log records

- [start transaction, T]: Start transaction marker, records that transaction T has started execution.
- [read item, T, X]: Records that transaction T has read the value of database item X.
- [write item, T, X, old value, new value]: Records that T has changed the value of database item X from old value to new value.

System Log

Log records (Cont.)

- [commit, T]: Commit transaction marker, records that transaction T has completed successfully, and confirms that its effect can be committed (recorded permanently) to the database.
- [abort, T]: Records that transaction T has been aborted.

System Log

Sample log

[start_transaction, T_1]
[read_item, T_1 , A]
[read_item, T_1 , D]
[write_item, T_1 , D, 20, 25]
[commit, T_1]
[checkpoint]
[start_transaction, T_2]
[read_item, T_2 , B]
[write_item, T_2 , B, 12, 18]
[start_transaction, T_4]
[read_item, T_4 , D]
[write_item, T_4 , D, 25, 15]
[start_transaction, T_3]
[write_item, T_3 , C, 30, 40]
[read_item, T_4 , A]
[write_item, T_4 , A, 30, 20]
[commit, T_4]
[read_item, T_2 , D]
[write_item, T_2 , D, 15, 25]

Transaction Roll Back

If a transaction fails for whatever reason after updating the database, but before the transaction commits, it may be necessary to **roll back** the transaction.

If any data item values have been changed by the transaction and written to the database, they must be restored to their previous values

Undo And Redo

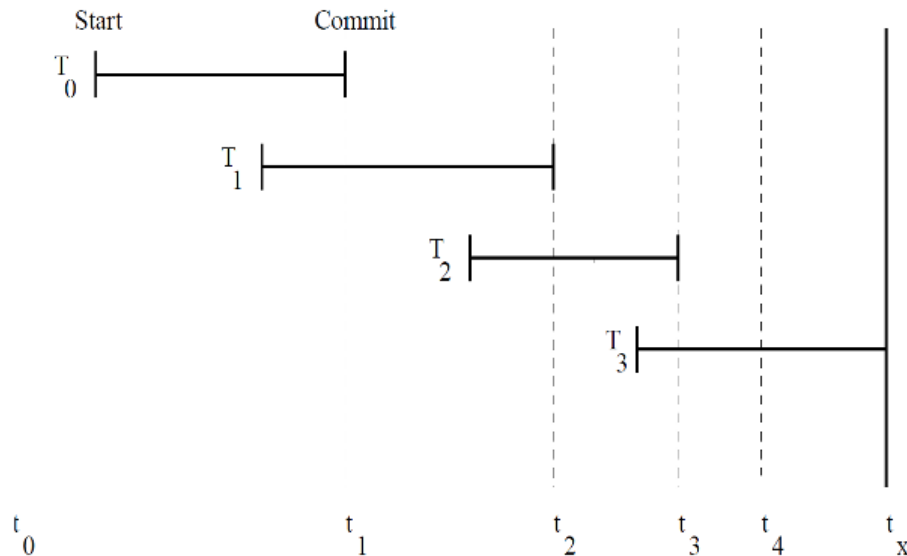
Procedure UNDO (WRITE_OP):

- Undoing a write_item operation WRITE_OP
- Consists of examining its log entry [write_item, T, X, old_value, new_value] and setting the value of item X in the database to old_value.
- Undoing a number of write_item operations from one or more transactions from the log must proceed in the reverse order from the order in which the operations were written in the log.

Procedure REDO (WRITE_OP):

- Redoing a write_item operation WRITE_OP
- Consists of examining its log entry [write_item, T, X, old_value, new_value] and setting the value of item X in the database to new_value.

Log-based Recovery



Assume that the database was recently created, the diagram shows transactions up until a crash.

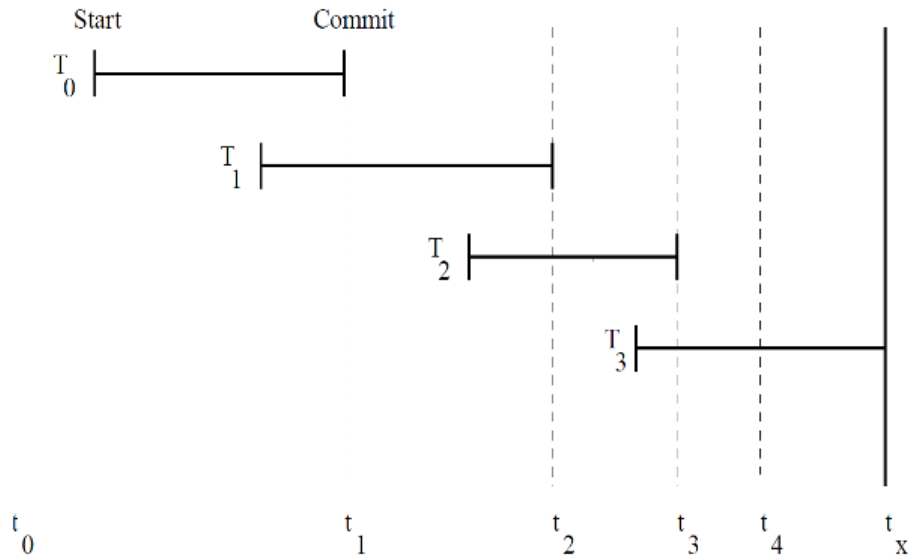
The database state will be somewhere between that at t_0 and the state at t_x (also true for log entries)

Write-Ahead Logging

Write-ahead log strategy

- Must **force** the **log record** for an update **before** corresponding data page gets to the disk.
- Must **force all log records** for a transaction **before** commit.

Log-based Recovery



Suppose the log was last written to disk at t_4 (shortly after t_3)

We would know that T_0 , T_1 and T_2 have committed, and their effects **should be** reflected in the database after recovery.

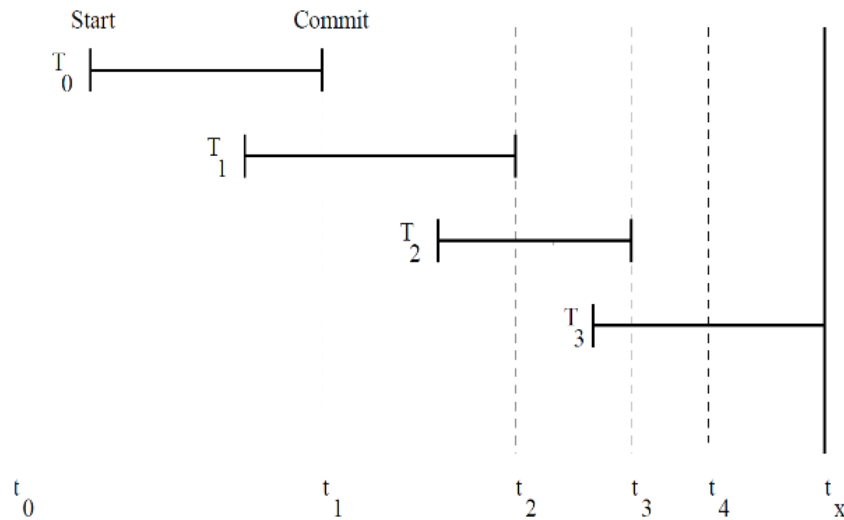
We also know that T_3 has started, may have modified some data, but is not committed. Thus, T_3 should be undone.

Rolling back (Undo) T_3

With a write-ahead strategy, we would be able to make some recovery by rolling back T_3

Step 1:

Undo the values written by T_3 to the old data values from the log



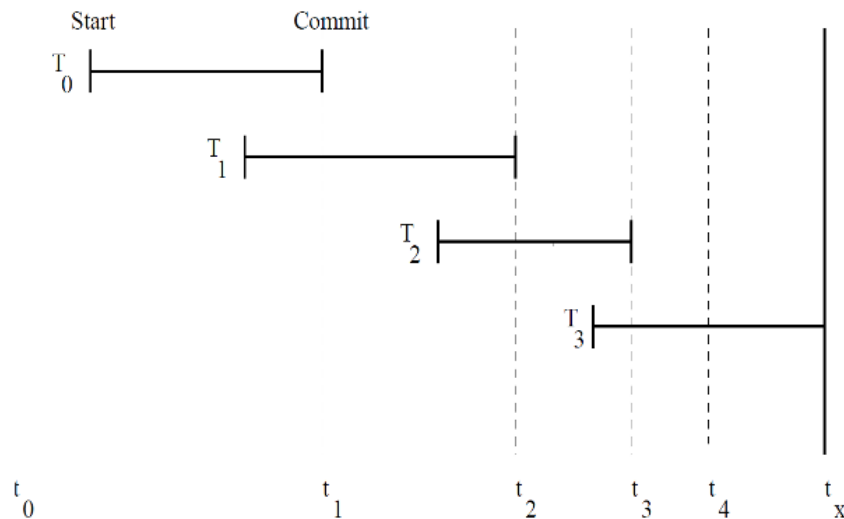
Undo helps guarantee **atomicity** in **ACID**

Redoing $T_0 \dots T_2$

With a write-ahead strategy, we would be able to make some recovery by rolling back T_3

Step 2:

Redoing the changes made by $T_0 \dots T_2$ using the new data values (for these committed transactions) from the log.



Redo guarantees **durability** in **ACID**

Checkpoints

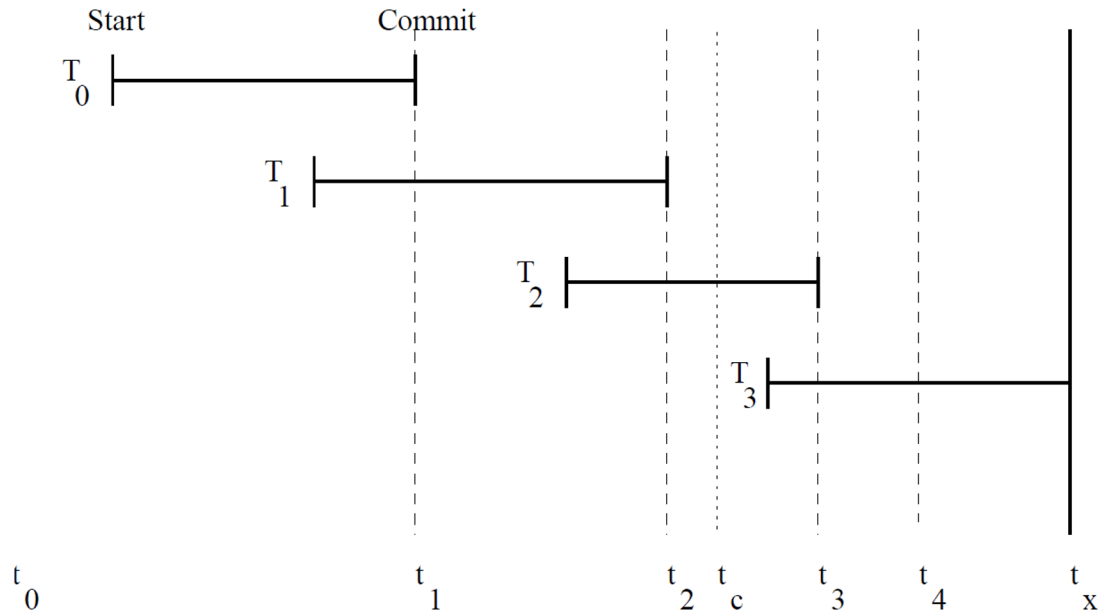
Notice also that using this system, the longer the time between crashes, the longer recovery may take.

To reduce this problem, the system could take **checkpoints** at regular intervals.

Taking a checkpoint consists of the following actions:

- Suspend execution of transactions temporarily.
- Force-write all main memory buffers that have been modified to disk.
- Write a [checkpoint] record to the log, and force-write the log to disk.
- Resume executing transactions.

Log Checkpoints



In our example, suppose a checkpoint is taken at time t_c . Then on recovery, we only need redo T_2 .

Catastrophic Failures

- **Disk failure.** Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash.
- **Physical problems and catastrophes.** This refers to an endless list of problems that includes power or fire, theft, overwriting disks by mistake...
- **Natural disaster.** Earthquake, flood, storms, tsunami...

Database Backup

The main technique used to handle such crashes is a **database backup**

- (1) whole database and (2) the log
- The latest backup copy can be reloaded, and the system can be restarted.

To not lose all transactions they have performed since the last database backup. We also backup the system log at more frequent intervals than full database backup.

Disaster Recovery

A plan for disaster recovery can include one or more of the following:

- A site to be used in the event of an emergency
- A different machine on which to recover the database
- Offsite storage of either database backups, table space backups, or both, as well as archived logs.

Learning Outcomes

Concurrency Control Techniques:

- Lock-based
- Ensures serializability, but could result in deadlocks

Database Recovery:

- Logs, Checkpoints, Backups