Transaction Management

Air-line Reservation

10 available seats vs 15 travel agents.

How do you design a robust and fair reservation system?

- Do not have enough resources
- Fair policy to every body
- Robustness

Failures

Number of factors might cause failures in user requirements processing.

1. System failure:
   - Disk failure - e.g. head crash, media fault.
   - System crash - unexpected failure requiring a reboot.
2. Program error - e.g. a divide by zero.
3. Exception conditions - e.g. no seats for your reservation.
4. Concurrency control - e.g. deadlock, expired locks.

To handle failures correctly and efficiently

Each database user must express his requirements as a set of program units.

Each program unit is a *transaction* that either

- accesses the contents of the database, or
- changes the state of the database, from one consistent state to another.

- Sydney → Tokyo → LA → N.Y
- It does not make sense only partial trip has tickets

Example transaction: buy a ticket from Sydney to N.Y. by JAL.

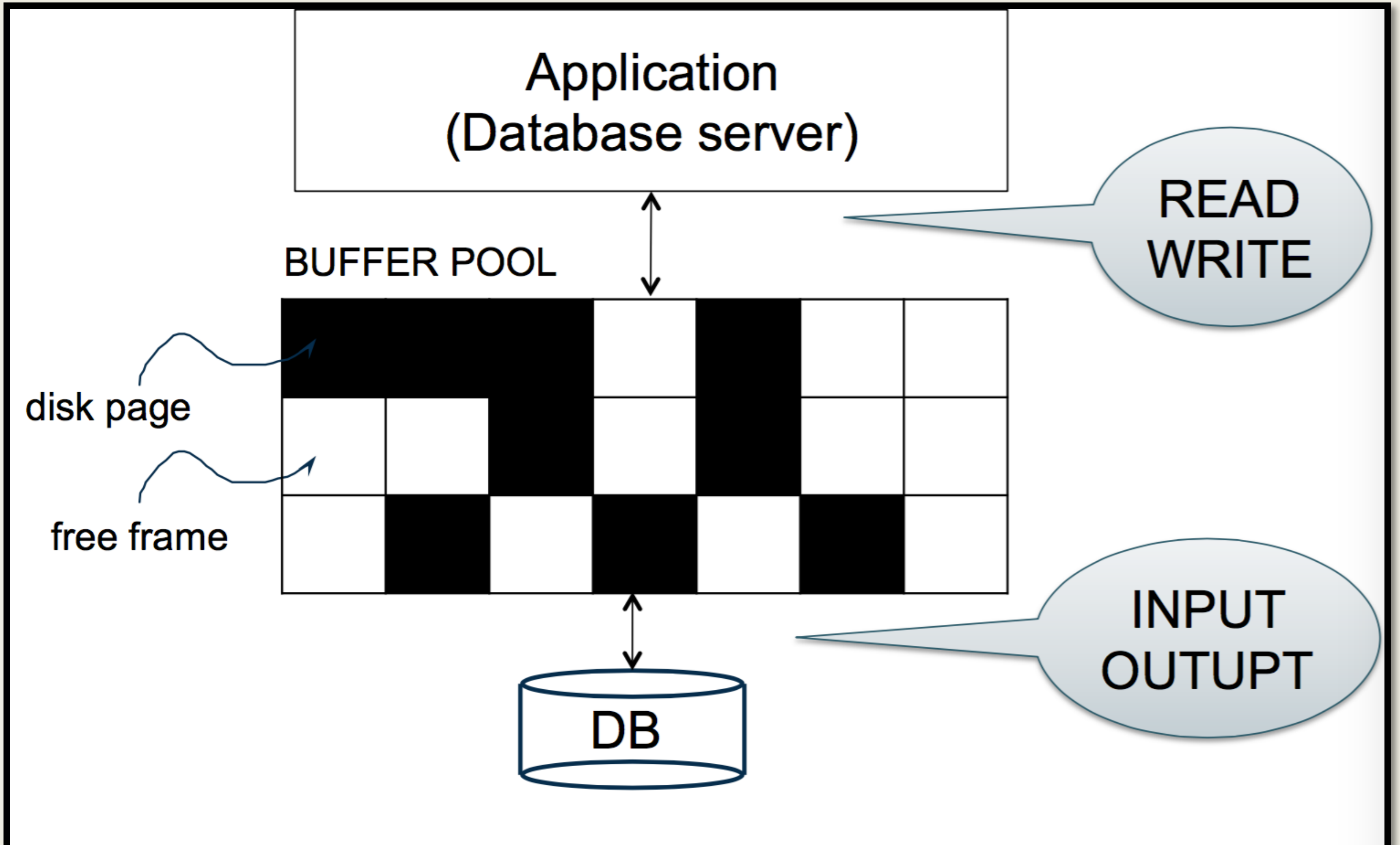
A transaction must be treated as an *atomic* unit.

Transaction

Three kinds of operations may be used in a transaction:

- *Read*.
- *Write*.
- Computation.

Buffer Management in a DBMS



Read

1. Compute the data block that contains the item to be read

2. Either
   - find a buffer containing the block, or
   - read from disk into a buffer

3. Copy the value from the buffer.

Write

1. Compute the disk block containing the item to be written

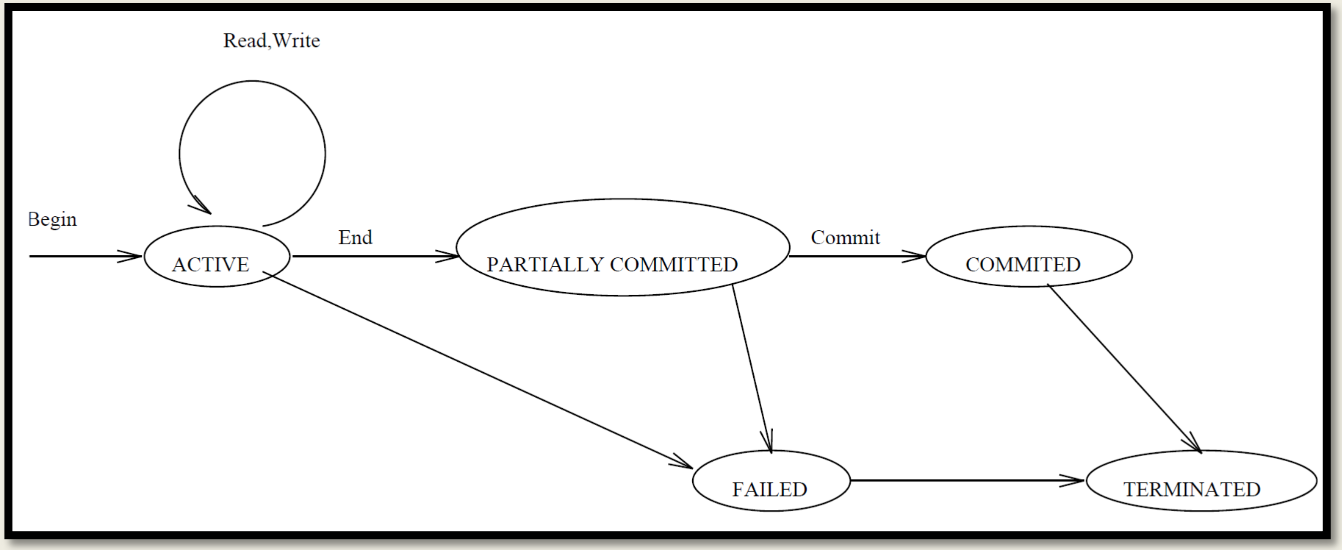
2. Either
   - find a buffer containing the block, or
   - read from disk into a buffer

3. Copy the new value into the buffer

4. At some point (maybe later), write the buffer back to disk.

Processing States of a Transaction

The typical processing states are illustrated in the figure below (E/N Fig 17.4):



Partially committed point: At this point, check and enforce the correctness of the concurrent execution.

Committed state: Once a transaction enters the committed state, it has concluded its execution successfully.

Desirable Properties of Transaction Processing **ACID**

- ***Atomicity***: A transaction is either performed in its entirety or not performed at all.
- ***Consistency preservation***: A correct execution of the transaction must take the database from one consistent state to another.
- ***Isolation***: A transaction should not make its updates visible to other transactions until it is committed.
- ***Durability or permanency***: Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

Problems without Enforcing ACID

- For a banking system,
  - ➢If durability is not enforced, then a customer may lose a deposit.
  - ➢If consistency preservation is not enforced, then the bank runs a high risk of bankrupt. E.g., run-over upper-limit.
- Below are the problems if atomicity and isolation are not enforced in a concurrent execution of transactions.

Lost Update Problem (Isolation is not enforced)

Suppose we have these two transactions, T_1 and T_2:

T_1:

read(X)
$X \leftarrow X + N$
write(X)
read(Y)
$Y \leftarrow Y - N$
write(Y)

T_2:

read(X)
$X \leftarrow X + M$
write(X)

- Let us see what may happen if T_1 and T_2 are executed concurrently in an uncontrolled way:

Suppose initially that $X = 100; Y = 50; N = 5$ and $M = 8$.

Database	T_1	T_2
$X = 100, Y = 50$	$X = ?, Y = ?$	$X = ?$
$X = 100, Y = 50$	read(X) $X = 100, Y = ?$	$X = ?$
$X = 100, Y = 50$	$X \leftarrow X + N$ $X = 105, Y = ?$	$X = ?$
$X = 100, Y = 50$	$X = 105, Y = ?$	read(X) $X = 100$
$X = 100, Y = 50$	$X = 105, Y = ?$	$X \leftarrow X + M$ $X = 108$
$X = 105, Y = 50$	write(X) $X = 105, Y = ?$	$X = 108$
$X = 105, Y = 50$	read(Y) $X = 105, Y = 50$	$X = 108$
$X = 108, Y = 50$	$X = 105, Y = 50$	write(X) $X = 108$
$X = 108, Y = 50$	$Y \leftarrow Y - N$ $X = 105, Y = 45$	$X = 108$
$X = 108, Y = 45$	write(Y) $X = 105, Y = 45$	$X = 108$

At the end of T_1 and T_2, X should be 113, Y should be 45.

The update $X \leftarrow X + N$ has been lost.

The Temporary Update Problem

Database	T_1	T_2
$X = 100, Y = 50$	$X = ?, Y = ?$	$X = ?$
$X = 100, Y = 50$	read(X) $X = 100, Y = ?$	$X = ?$
$X = 100, Y = 50$	$X \leftarrow X + N$ $X = 105, Y = ?$	$X = ?$
$X = 105, Y = 50$	write(X) $X = 105, Y = ?$	$X = ?$
$X = 105, Y = 50$	**FAILS**	read(X) $X = 105$
$X = 105, Y = 50$		$X \leftarrow X + M$ $X = 113$

Recover from the disk

Several possibilities for what might happen next:

Case 1:

Database	T_1	T_2
X = 105, Y = 50	**DBMS undoes T_1**	X = 113
X= 100, Y = 50		X = 113
X=113, Y=50		Write (X) X= 113

Case 2:

Database	T_1	T_2
X =105, Y = 50		X = 113
X= 105, Y = 50	**DBMS does nothing to T_1**	X = 113
X=113, Y=50		Write (X) X= 113

Case 3:

Database	T_1	T_2
X = 105, Y = 50		X = 113
X= 105, Y = 50		X = 113
	DBMS undoes T_1	Write (X), X= 113
X=100, Y=50		X = 100

In case 1 and 2, only half of T_1 has been executed. In case 3, T_2 has been lost.

The Incorrect Summary Problem

T_1	T_3
	$sum \leftarrow 0$
	read(A)
	$sum \leftarrow sum + A$
	$\vdots$
read(X)	
$X \leftarrow X - N$	
write(X)	
	$\vdots$
	read(X)
	$sum \leftarrow sum + X$
	read(Y)
	$sum \leftarrow sum + Y$
	$\vdots$
read(Y)	
$Y \leftarrow Y + N$	
write(Y)	
	$\vdots$

Here the sum calculated by T_3 will be wrong by N.

Recover from Failures

Ensure the A in ACID

Log-based Recovery

- Undo logging
- Redo logging
- Undo/Redo logging

System Log

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

System Log

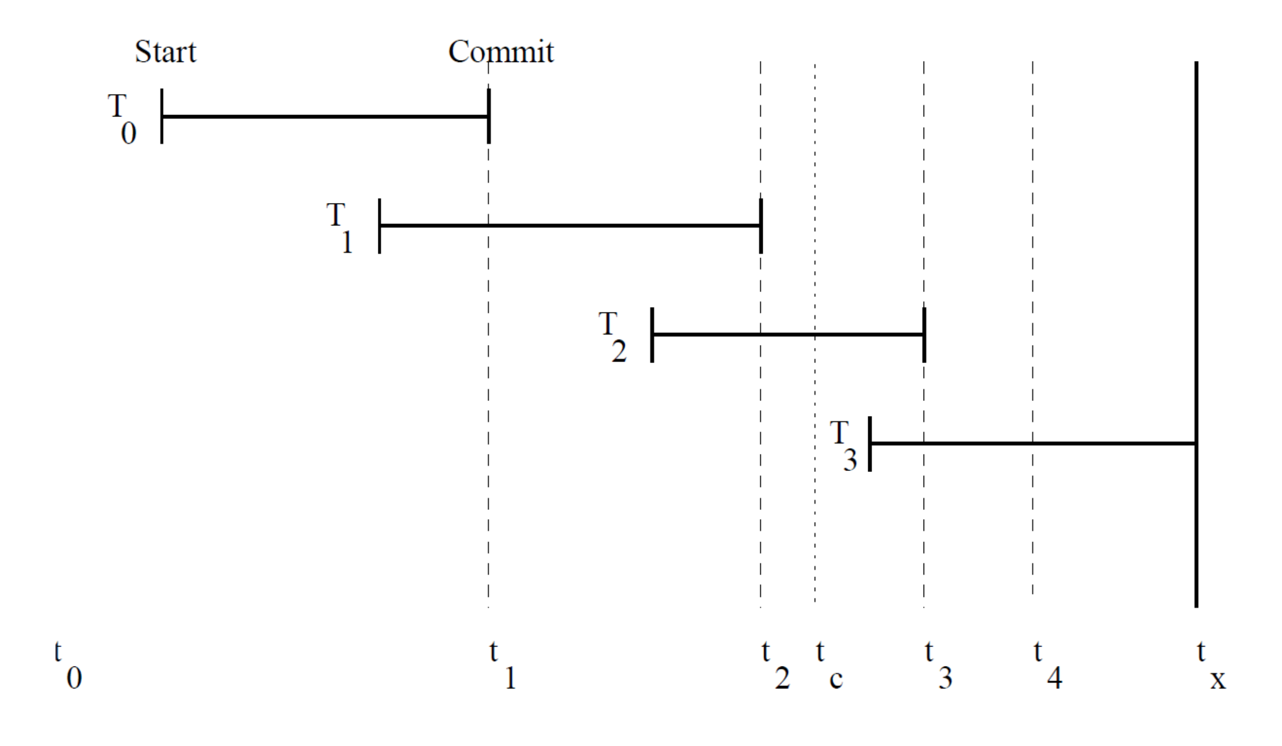
- Start transaction marker [start transaction, *T*]: Records that transaction *T* has started execution.
- [read item, *T*, *X*]: Records that transaction *T* has read the value of database item *X*.
- [write item, *T*, *X*, old value, new value]: Records that *T* has changed the value of database item *X* from old value to new value.
- Commit transaction marker [commit, *T*]: 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 (Cont'd)

In fact some other entries (rollback, undo, redo) are also required for a recovery method.

These entries allow the recovery manager to *rollback* an unsuccessful transaction (undo any partial updates).

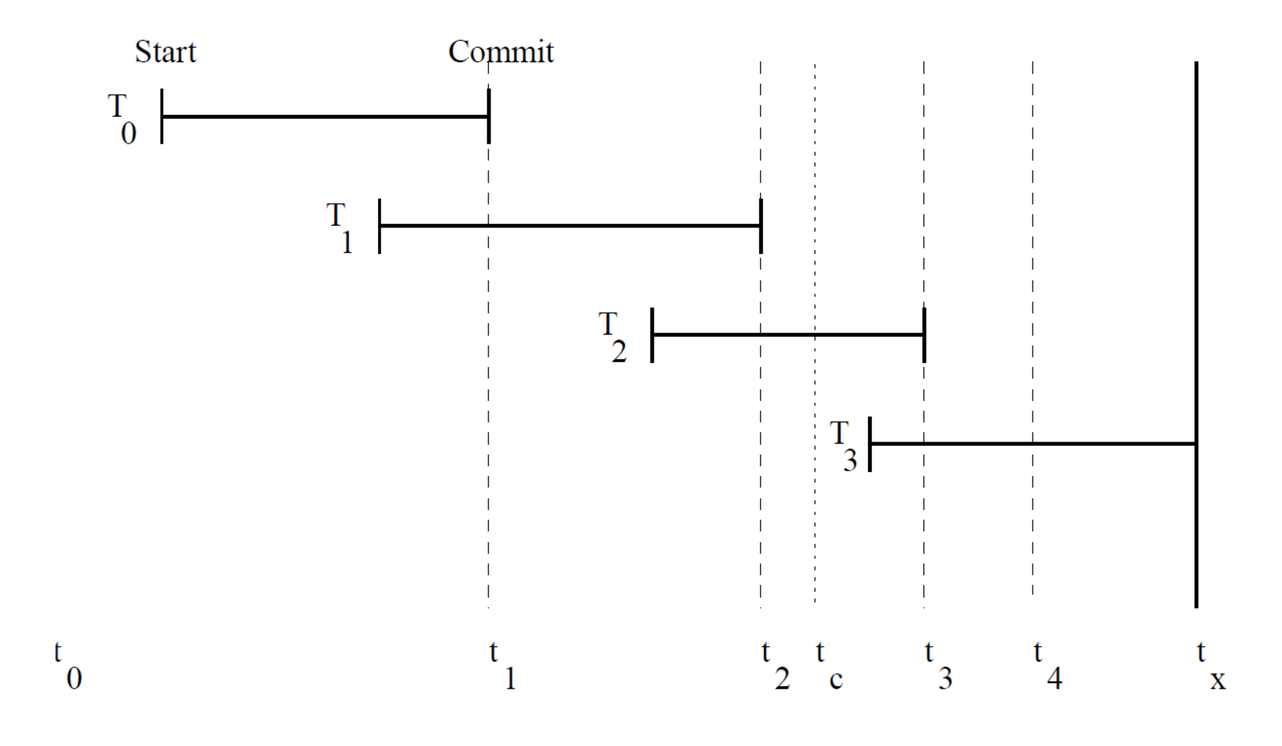
Recovery



Let us see how the log might be used to recover from a system crash.

The diagram below shows transactions between the last system backup and a crash.

Recovery (Cont'd)



The database on disk will be in a state somewhere between that at t_0 and the state at t_x.x

The same is also true for log entries.

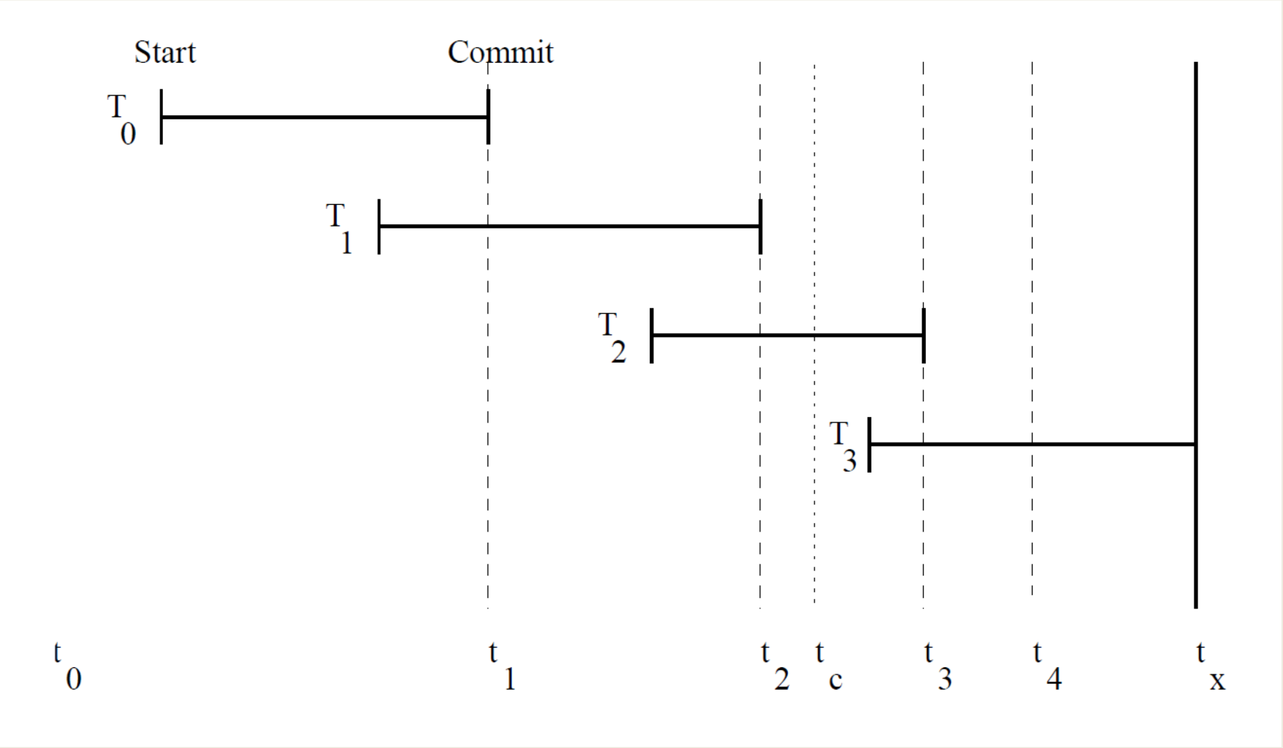
Recovery (Cont'd)

We will assume that the ***write-ahead log strategy*** is used. This means that

- old data values must be force-written to the log (i.e. the buffer must be copied to disk) before any change can be made to the database, and

- the transaction is regarded as committed when the new data values and the commit marker have been force-written to the log.

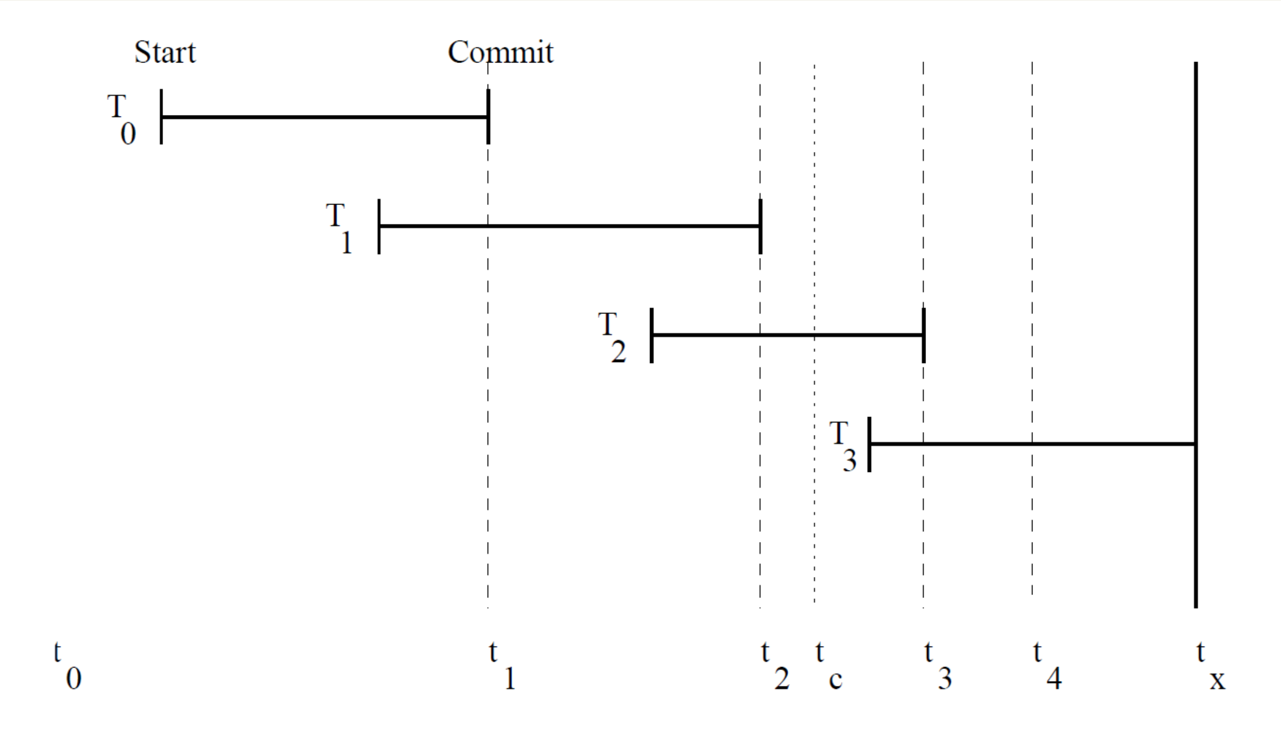
Thus the log is force-written at least at t_1, t_2 and t_3 in the above.



Suppose the log was last written to disk at t_4.

By examining the log:

1. We know that T_0, T_1 and T_2 have committed and their effects should be reflected in the database after recovery.
2. But we do not know whether the effects of T_0, T_1 and T_2 were reflected at the time of the crash.
3. We also know that T_3 has started, may have modified some data, but is not committed. Thus T_3 should be undone.



The database can be recovered by rolling back T_3 using the old data values from the log, and redoing the changes made by T_0 … T_2 using the new data values (for these committed transactions) from the log.

Notice that instead of rolling back, the database could have been restored from the backup. This might be necessary in the event of a disk crash for example (for this reason, the log should be stored on an independent disk pack).

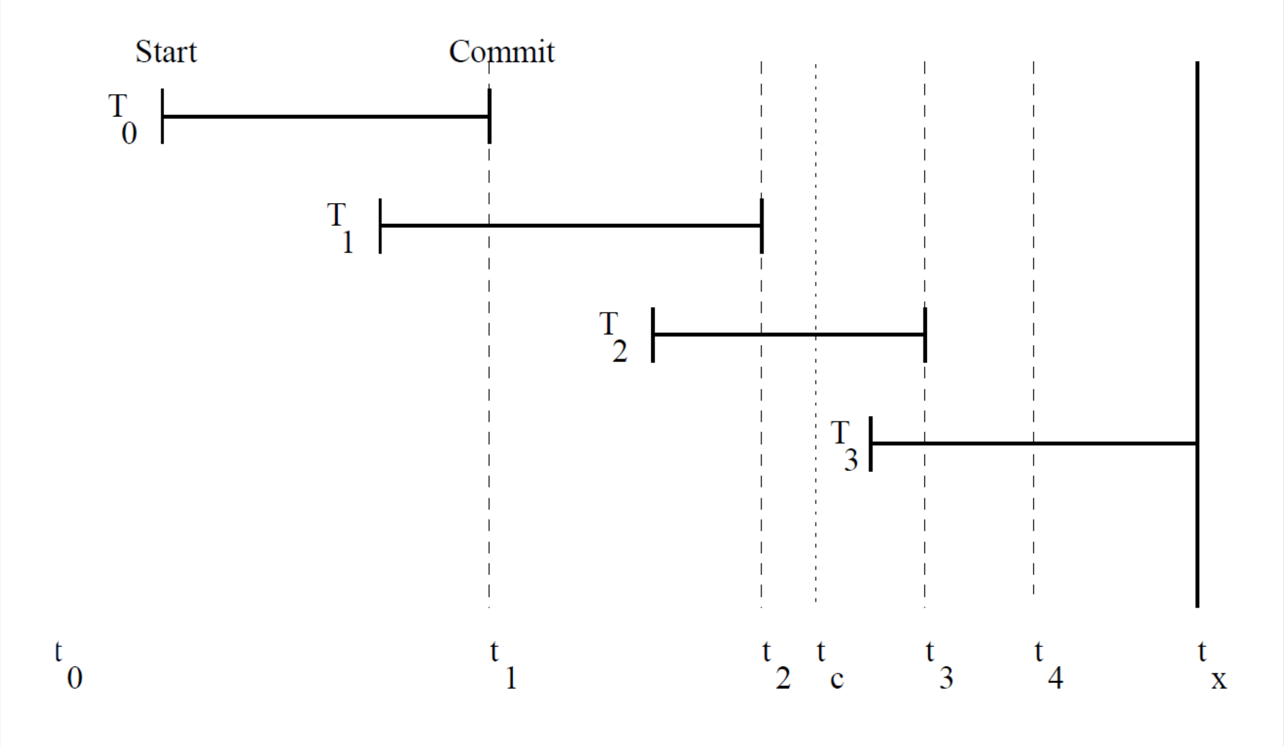
Checkpoints

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

To avoid this problem, the system may take *checkpoints* at regular intervals.

To do this:

- a *start of checkpoint* marker is written to the log, then
- the database updates in buffers are force-written, then
- an *end of checkpoint* marker is written to the log.



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

Recall: Desirable Properties of Transaction Processing: **ACID**

- ***Atomicity***: A transaction is either performed in its entirety or not performed at all.
- ***Consistency preservation***: A correct execution of the transaction must take the database from one consistent state to another.
- ***Isolation***: A transaction should not make its updates visible to other transactions until it is committed.
- ***Durability or permanency***: Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

Concurrency Control

- Multiple concurrent transactions T_1, T_2, …
- They read/write common elements A_1, A_2, …
- How can we prevent unwanted interference?

The Scheduler is responsible for that

Schedules of Transactions

To fully utilise resources, desirable to interleave the operations of transactions in an appropriate way.

For example, if one transaction is waiting for I/O to complete, another transaction can use the CPU.

A *schedule S* of the transactions $T_1, \ldots, T_n$

- is a sequential ordering of the operations of $T_1, \ldots, T_n$, and
- preserves the ordering of operations in each transaction T_i.

Example Schedules

(a)

T_1	T_2
read(X)	
$X \leftarrow X + N$	
write(X)	
read(Y)	
$Y \leftarrow Y - N$	
write(Y)	
	read(X)
	$X \leftarrow X + M$
	write(X)

(b)

T_1	T_2
	read(X)
	$X \leftarrow X + M$
	write(X)
read(X)	
$X \leftarrow X + N$	
write(X)	
read(Y)	
$Y \leftarrow Y - N$	
write(Y)	

Example Schedules (Cont.)

(c)

T_1	T_2
read(X)	
$X \leftarrow X + N$	
	read(X)
	$X \leftarrow X + M$
write(X)	
read(Y)	
	write(X)
$Y \leftarrow Y - N$	
write(Y)	

(d)

T_1	T_2
read(X)	
$X \leftarrow X + N$	
write(X)	
	read(X)
	$X \leftarrow X + M$
	write(X)
read(Y)	
$Y \leftarrow Y - N$	
write(Y)	

Serial Schedule

As we have seen, if operations are interleaved arbitrarily, incorrect results may occur.

However, it is reasonable to assume that schedules (a) and (b) in the figure will give correct results (as long as the transactions are independent).

(a) and (b) are called *serial* schedules, and we will assume that *any serial schedule is correct.*

(a)

T_1	T_2
read(X)	
$X \leftarrow X + N$	
write(X)	
read(Y)	
$Y \leftarrow Y - N$	
write(Y)	
	read(X)
	$X \leftarrow X + M$
	write(X)

(b)

T_1	T_2
	read(X)
	$X \leftarrow X + M$
	write(X)
read(X)	
$X \leftarrow X + N$	
write(X)	
read(Y)	
$Y \leftarrow Y - N$	
write(Y)	

Serializable Schedule

Notice that schedule (d) always produces the same result as schedule (a), so it should also give correct results.

A schedule is *serializable* if it always produces the same result as *some* serial schedule.

Notice that schedule (c) is not serializable.

(c)

T_1	T_2
read(X)	
$X \leftarrow X + N$	
	read(X)
	$X \leftarrow X + M$
write(X)	
read(Y)	
	write(X)
$Y \leftarrow Y - N$	
write(Y)	

(d)

T_1	T_2
read(X)	
$X \leftarrow X + N$	
write(X)	
	read(X)
	$X \leftarrow X + M$
	write(X)
read(Y)	
$Y \leftarrow Y - N$	
write(Y)	

Conflict Serializable Schedules

Two schedules are ***conflict equivalent*** if:

- Involve the same actions of the same transactions
- Every pair of conflicting actions is ordered the same way

Two operations O_1 and O_2 are *conflicting* if

- they are in different transactions but on the same data item,
- one of them must be a write.

Schedule S is ***conflict serializable*** if S is conflict equivalent to some serial schedule

Conflict Serializability

Any conflict serializable schedule is also a serializable schedule

The inverse is not true.

Testing Conflict Serializable

Why not run only serial schedules? That is, run one transaction after the other?

Because of very poor throughput due to disk latency

When there are only two transactions, there are only two serial schedules - for n transactions there will be $n!$.

Fortunately there is an efficient algorithm to check whether a schedule is conflict serializable without checking all these possibilities.

Check Conflict Serializability

Algorithm

Step 1: Construct a *schedule* (or *precedence*) graph – a *directed graph*.

Step 2: Check if the graph is *cyclic*:

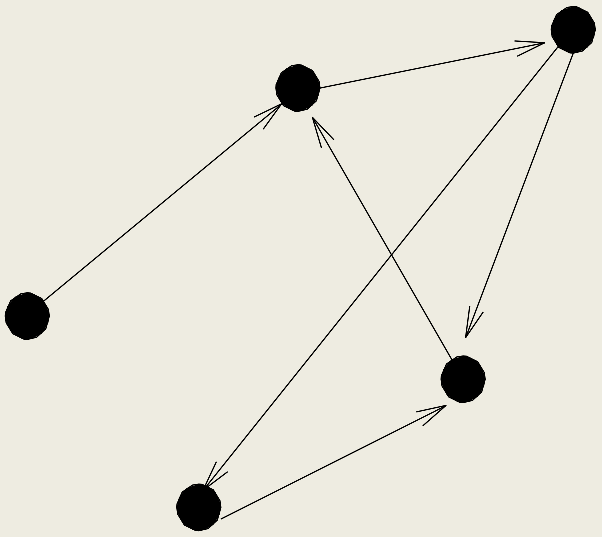
- Cyclic: non-serializable.
- Acyclic: serializable.

Definitions

A *directed graph* $G = (V, A)$ consists of

- a vertex set V, and
- an arc set A such that each arc connects two vertices.

G is *cyclic* if G contains a directed cycle.



Cyclic Graph

Construct a Schedule Graph $G_S = (V, A)$ for a schedule S

1. A vertex in V represents a transaction.

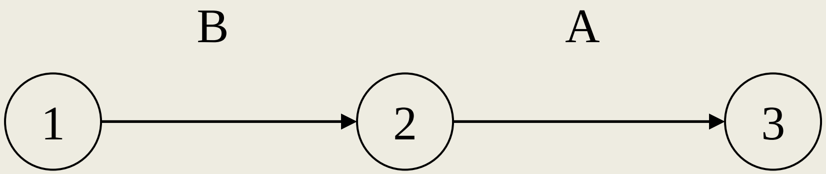
2. For two vertices T_i and T_j, an arc $T_i \rightarrow T_j$ is added to A if
   - there are two *conflicting* operations $O_1 \in T_i$ and $O_2 \in T_j$,
   - in S, O_1 is before O_2.

Two operations O_1 and O_2 are *conflicting* if

- they are in different transactions but on the same data item,
- one of them must be a write.

Example 1

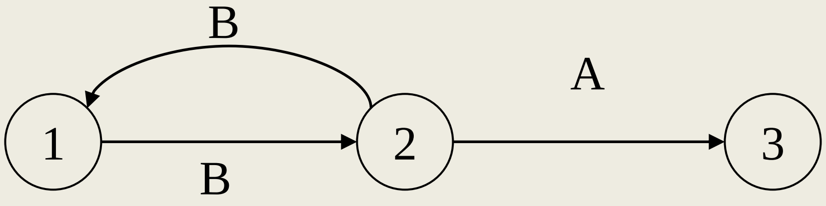
$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$



This schedule is conflict-serializable

Example 2

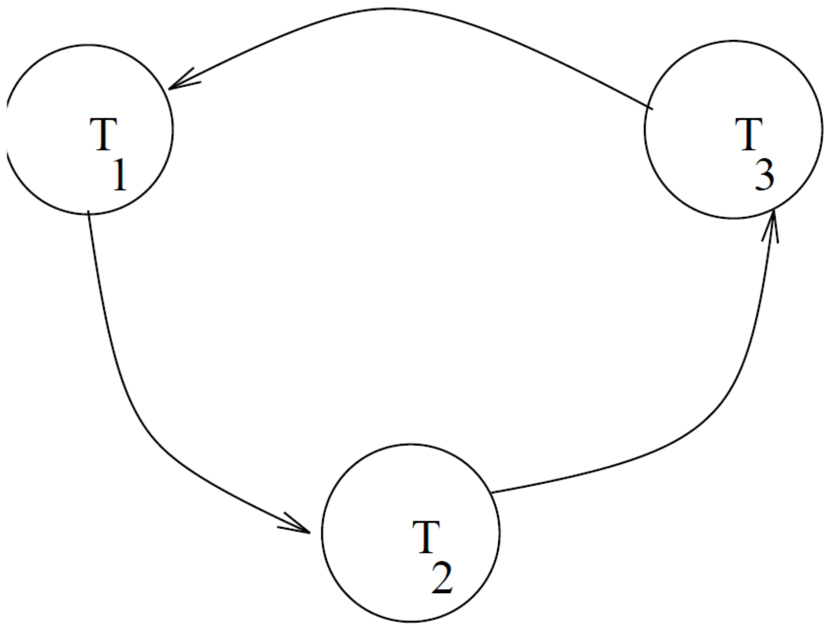
$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$



This schedule is NOT conflict-serializable

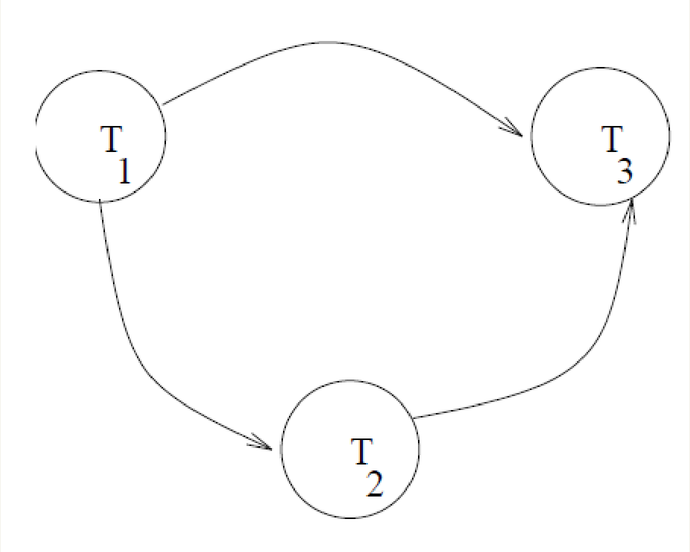
Example 1:

Schedule	T_1	T_2	T_3
read(A)	read(A)		
read(B)		read(B)	
$A \leftarrow f_1(A)$	$A \leftarrow f_1(A)$		
read(C)			read(C)
$B \leftarrow f_2(B)$		$B \leftarrow f_2(B)$	
write(B)		write(B)	
$C \leftarrow f_3(C)$			$C \leftarrow f_3(C)$
write(C)			write(C)
write(A)	write(A)		
read(B)			read(B)
read(A)		read(A)	
$A \leftarrow f_4(A)$		$A \leftarrow f_4(A)$	
read(C)	read(C)		
write(A)		write(A)	
$C \leftarrow f_5(C)$	$C \leftarrow f_5(C)$		
write(C)	write(C)		
$B \leftarrow f_6(B)$			$B \leftarrow f_6(B)$
write(B)			write(B)


T_1
T_3
T_2

Example 2:

Schedule	T_1	T_2	T_3
read(A)	read(A)		
$A \leftarrow f_1(A)$	$A \leftarrow f_1(A)$		
read(C)	read(C)		
write(A)	write(A)		
$A \leftarrow f_2(C)$	$A \leftarrow f_2(C)$		
read(B)		read(B)	
write(C)	write(C)		
read(A)		read(A)	
read(C)			read(C)
$B \leftarrow f_3(B)$		$B \leftarrow f_3(B)$	
write(B)		write(B)	
$C \leftarrow f_4(C)$			$C \leftarrow f_4(C)$
read(B)			read(B)
write(C)			write(C)
$A \leftarrow f_5(A)$		$A \leftarrow f_5(A)$	
write(A)		write(A)	
$B \leftarrow f_6(B)$			$B \leftarrow f_6(B)$
write(B)			write(B)



Unfortunately, testing for serializability on the fly is not practical.

Instead, a number of protocols have been developed which ensure that if every transaction obeys the rules, then *every* schedule will be serializable, and thus correct.

Concurrency Control Methods

Locking Mechanism

The idea of locking some data item *X* is to:

- give a transaction exclusive use of the data item *X*,
- do not restrict the access of other data items.

This prevents one transaction from changing a data item currently being used in another transaction.

We will discuss a simple locking scheme which locks individual items, using read and write locks

Locking Rules

In this schema, every transaction *T* must obey the following rules.

1) If *T* has only one operation (read/write) manipulating an item *X*:

- obtain a read lock on *X* before reading it,
- obtain a write lock on *X* before writing it,
- unlock *X* when done with it.

2) If *T* has several operations manipulating *X*:

- obtain one proper lock only on *X*:

  a read lock if all operations on *X* are reads;

  a write lock if one of these operations on *X* is a write.
- unlock *X* after the last operation on *X* in *T* has been executed.

Locking Rules (cont.)

In this scheme,

- Several read locks can be issued on the same data item at the same time.

- A read lock and a write lock cannot be issued on the same data item at the same time, neither two write locks.

This still does not guarantee serializability.

Example: Based on E/N Fig 18.3.

T_1	T_2
read_lock(Y)	
read(Y)	
unlock(Y)	
	read_lock(X)
	read(X)
	unlock(X)
	write_lock(Y)
	read(Y)
	$Y \leftarrow X + Y$
	write(Y)
	unlock(Y)
write_lock(X)	
read(X)	
$X \leftarrow X + Y$	
write(X)	
unlock(X)	

Two Phase Locking (2PL)

To guarantee serializability, transactions must also obey the *two-phase locking protocol*:

- ***Growing Phase***: all locks for a transaction must be obtained before any locks are released, and

- ***Shrinking Phase***: gradually release all locks (once a lock is released no new locks may be requested).

Two Phase Locking (2PL) (Cont.)

Example: Based on E/N Fig 18.4.

T_1
read_lock(Y)
read(Y)
write_lock(X)
unlock(Y)
read(X)
$X \leftarrow X + Y$
write(X)
unlock(X)

Locking thus provides a solution to the problem of correctness of schedules.

Two phase locking ensures conflict serializability

Deadlock

A problem that arises with locking is **deadlock**.

Deadlock occurs when two transactions are each waiting for a lock on an item held by the other.

T_1	T_2
write_lock(X)	
read(X)	
	write_lock(Y)
	read(Y)
write_lock(Y)	
waiting for Y*	write_lock(X)
waiting for Y*	***waiting for X***

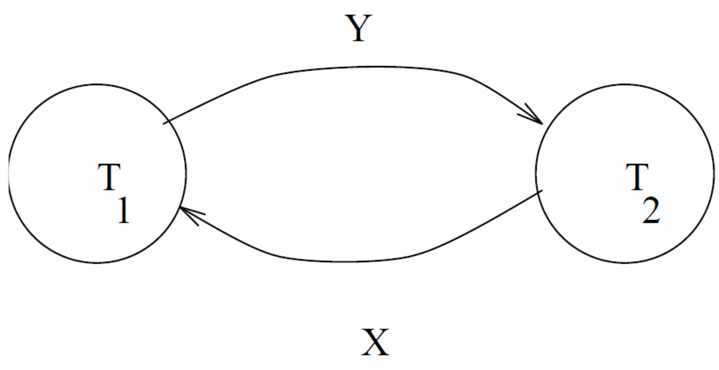
Deadlock Check

Create the *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:



Several methods to deal with deadlocks

deadlock detection

- periodically check for deadlocks, abort and rollback some transactions (restart them later). This is a good choice if transactions are very short or very independent.

Several methods to deal with deadlocks (Cont.)

<u>deadlock prevention</u> - Assign priorities based on timestamps. Assume T_i wants a lock that T_j holds. Two policies are possible:

- Wait-Die: If T_i has higher priority, T_i waits for T_j; otherwise T_i aborts
- Wound-wait: If T_i has higher priority, T_j aborts; otherwise T_i waits

If a transaction re-starts, make sure it has its original timestamp

Timestamp ordering

The idea here is:

- to assign each transaction a timestamp (e.g. start time of transaction), and
- to ensure that the schedule used is equivalent to executing the transactions in timestamp order

Timestamp ordering

Each data item, X, is assigned

- a read timestamp, *read TS(X)* - the latest timestamp of a transaction that read X, and
- a write timestamp, *write TS(X)* - the latest timestamp of a transaction that write X.

Timestamp ordering

These are used in read and write operations as follows. Suppose the transaction timestamp is *T*.

$read(X)$:

```
If T >= write_TS(X) then
    { execute read(X);
    if T >= read_TS(X) then
          read_TS(X) <- T }
else
   rollback the transaction and restart
```

$write(X)$:

```
If T >= read_TS(X) and T >= write_TS(X) then
   { execute write(X); write_TS(X) <- T }
else
   rollback and restart
```

Tomas' Write Rule:

$write(X)$:

```
If T < read_TS(X) then
   rollback and restart
else if T < write_TS(X) then
        ignore the write
     else
        { execute write(X);
          write_TS(X) <- T }
```

Some Problems:

- **Cyclic restart**: There is no deadlock, but a kind of livelock can occur - some transactions may be constantly aborted and restarted.

- **Cascading rollback**: When a transaction is rolled back, so are any transactions which read a value written by it, and any transactions which read a value written by them . . . etc. This can be avoided by not allowing transactions to read values written by uncommitted transactions (make them wait).

Multiversioning

Similar to the timestamp ordering approach; but is allowed to access “old” versions of a table.

A history of the values and timestamps (versions) of each item is kept.

When the value of an item is needed, the system chooses a **proper** version of the item that maintains serializability.

This results in fewer aborted transactions at the cost of greater complexity to maintain more versions of each item.

Multiversioning

We will look at a scheme, several versions $X_1, \ldots, X_k$ of each data item are kept. For each X_i we also keep

- *read* $TS(X_i)$ - as for timestamp ordering.
- *write* $TS(X_i)$ - as for timestamp ordering.

Multiversioning

Read and write are done as follows for a transaction *P* with timestamp T.

$read(X)$:

```
Find Xi s.t. write_TS(Xi) is the
   highest write timestamp but <= T
update read_TS(Xi) (and do read(Xi))
return Xi as the value for X
```

Multiversioning

$write(X)$:

```
Find Xi s.t. write_TS(Xi) is the
   highest write timestamp but <= T
if T < read_TS(Xi) then
   rollback and restart
else
   { create a new version X(k+1) of X;
     set read_TS(X(k+1)) to T;
     set write_TS(X(k+1)) to T}
```

Note: Cascading rollback and cyclic restart problems can still occur, but should be reduced.

However, there is an increased overhead in maintaining multiple versions of items.

Optimistic scheduling

In two-phase locking, timestamp ordering, and multiversioning concurrency control techniques, a certain degree of checking is done **before** a database operation can be executed.

The idea here is to push on and hope for the best!

No checking is done while the transaction is executing.

The Protocol has Three Phases

- ***read phase*** - A transaction can read data items from the database into local variables. However, updates are applied only to local copies of the data items kept in the transaction workspace.

- ***validation phase*** - checks are made to ensure that serializability is not violated,

- ***write phase*** -if validation succeeds, updates are applied and the transaction is committed. Otherwise, the updates are discarded and the transaction is restarted.

A scheme uses timestamps and keeps each transaction's

- read-set - the set of items read by the transaction,
- write-set - the set of items written by the transaction.

During validation, we check that the transaction does not interfere with any transaction that is committed or currently validating.

Each transaction T is assigned 3 timestamps:

$Start(T)$, $Validation(T)$, $Finish(T)$.

To pass the validation test for T, one of the following must be true:

- $Finish(S) < Start(T)$; or
- for S s.t. $Start(T) < Finish(S)$, then
    a) write set of S is disjoint from the read set of T, and
    b) $Finish(S) < Validation(T)$.

Optimistic control is a good option if there is not much interaction between transactions.

2PL vs. TSO vs. MV vs. OP

A Comparison among two-phase locking (2PL), timestamp ordering (TSO), multiversioning (MV), optimistic (OP) concurrency control techniques.

MV should provide the greatest concurrency degree (in average). However, we need to maintain multiversions for each data item.

2PL can offer the second greatest concurrency degree (in average); but will result in deadlocks. To resolve the deadlocks, either

- need additional computation to detect deadlocks and to resolve the deadlocks, or
- reduce the concurrency degree to prevent deadlocks by adding other restrictions.

2PL vs. TSO vs. MV vs. OP (cont.)

If most transactions are very short, we can use 2PL + deadlock detection and resolution.

TSO has a less concurrency degree than that of 2PL if a proper deadlock resolution is found. However, TSO does not cause deadlocks. Other problems, such as cyclic restart and cascading rollback, will appear in TSO.

If there are not much interaction between transactions, OP is a very good choice. Otherwise, OP is a bad choice.

Understanding of MV and OP is optional in this course.

Learning Outcomes

- Recovery using system logs
- Serial, serializable, and conflict serializable schedules
- Testing conflict serializable by constructing a schedule (precedence) graph
- Lock mechanisms: simple locking scheme, 2 phase locking (2PL), and timestamp ordering