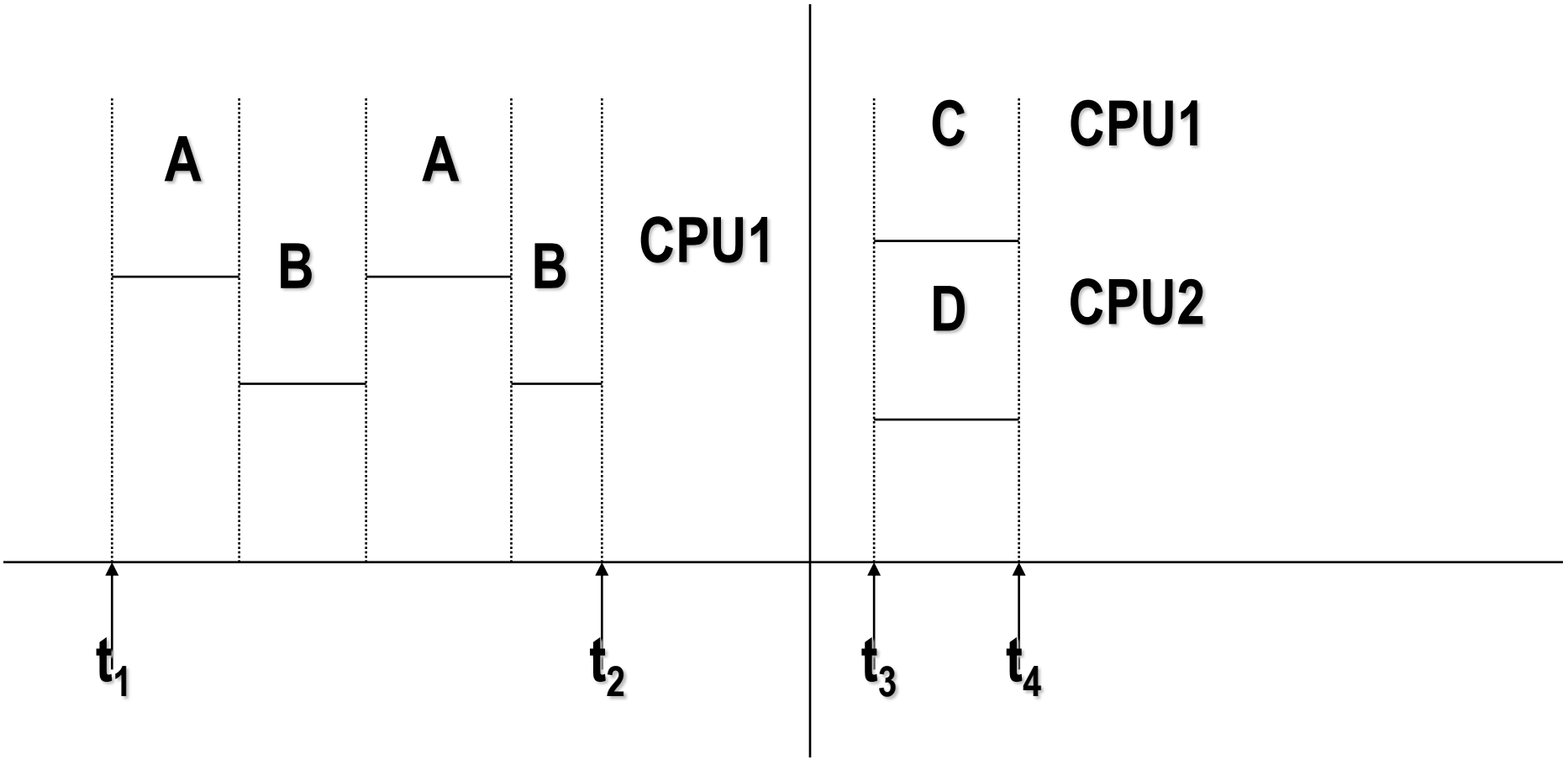
Transaction Processing, Serializability Theory & Concurrency Control, and Recovery

Introduction

Single-User Versus Mutiuser Systems



Introduction

Multiple users can use computer systems simultaneously because of multiprogramming.

This gives rise to interleaved execution of the programs with disk accesses and CPU processing.

With multiple CPUs simultaneous processing of multiple programs is possible.

The execution of a program that accesses or changes the contents of a database is called as transaction.

We are concerned with only interleaved execution of programs for transaction management.

Operations of a Transaction

A transaction accesses or modifies the contents of a database.

A database is treated as a set of data items and disk blocks which are accessed by transactions

- read_item(X) or R(X): Reads a database item X into a program variable.
  - –Find the address of the disk block that contains item X
  - –Copy the disk block in main memory if required
  - –Copy item X from the buffer to the program variable named X

A transaction with only read operations is known as read-only transaction and are not of much interest.

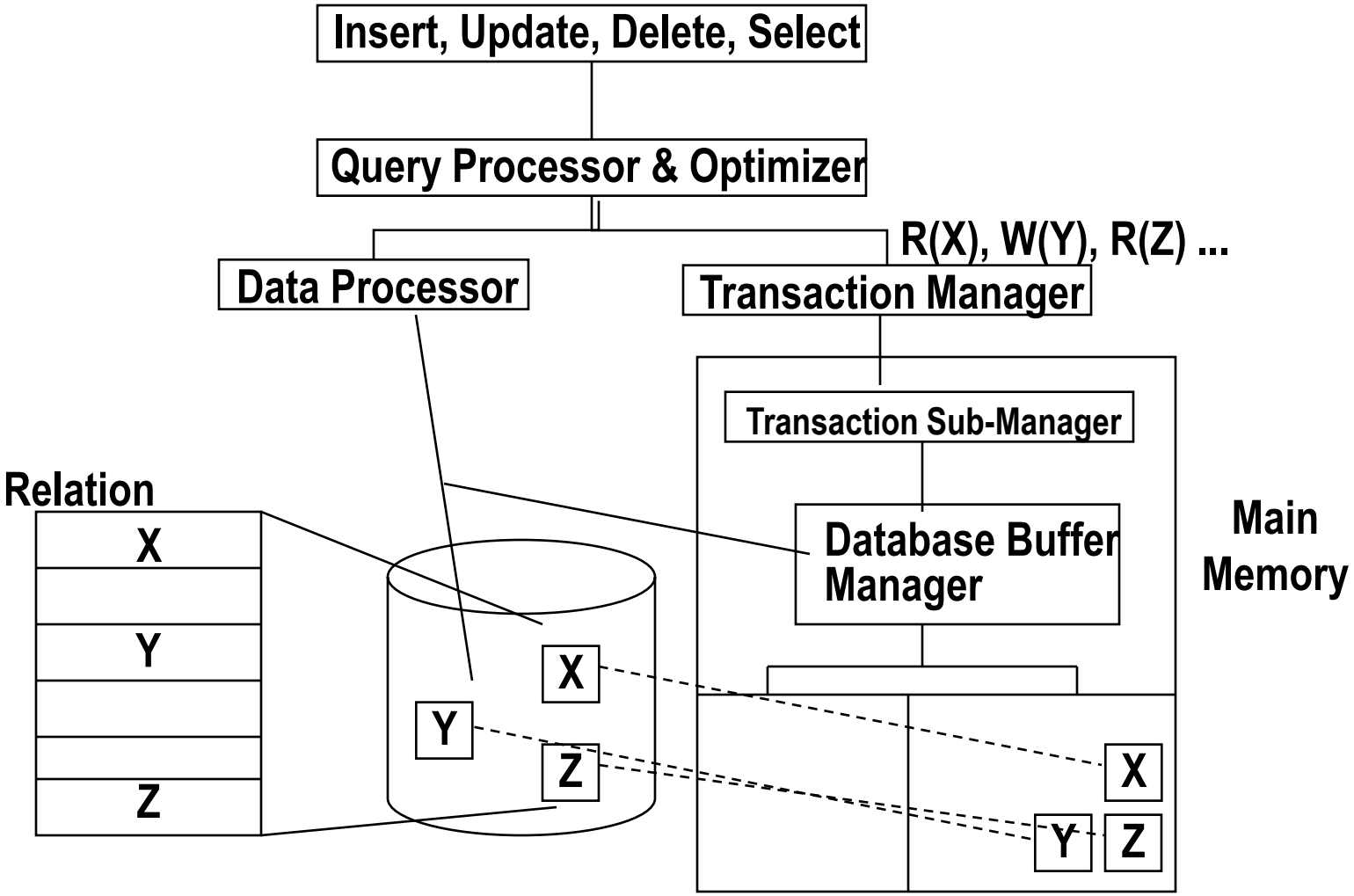
Operations of a Transaction (Cont.)

- write_item(X) or W(X): Writes the value of a program variable X into the database item named X
  - Find the address of the disk block that contains item X
  - Copy the disk block in main memory if required
  - Copy item X from the program variable named X into its correct location in the buffer
  - Store the updated block from the buffer back to disk (either immediately or at some later point in time)

Operations of a Transaction (Cont.)

- Additional operations of a transaction (which can be explicitly specified)
  - Commit - the transaction is successful and the data items value must be changed (if any)
  - Rollback/Abort - the transaction is not successful do not change any of the data item values

Conceptual View of Transaction Management



Why do you need the concept of transaction?

- to maintain database consistency over time (atomicity, isolation)
- to evaluate if multiple users can simultaneously access and modify the data (concurrency)
- to make the changes to data permanent (durability)

Why is Concurrency Control needed?

- Several problems occur when concurrent transactions execute in an uncontrolled manner
- In our examples illustrating the problems without concurrency control each execution of concurrent transactions is a particular interleaving of their read or write operations on data items.
- In general a transaction, has a set of data items it accesses (read set), and a set of data items it modifies (write set)

Lost Update Problem

A transaction *overwrites a data item modified* by other transactions

Account(Name, Bal) **tuple(Kiran, 1000)**

Kolkota ATM Kiran T1 Juhu ATM Aamir T2

R1(Bal)
Bal = Bal +500
W1(Bal)
EOT1

R2(Bal)
Bal = Bal - 700
W2(Bal)
EOT2

Schedule1	Bal
R1(Bal)	1000
R2(Bal)	1000
W1(Bal)	1500
W2(Bal)	300
EOT1	
EOT2	

Schedule2	Bal
R1(Bal)	1000
R2(Bal)	1000
W2(Bal)	300
W1(Bal)	1500
EOT1	
EOT2	

For a consistent state the value of balance, after execution of T1 and T2 should be 800

Temporary Update or Dirty Read

A transaction *reads uncommitted modified data item values* updated by other transactions.

Account(Name, Bal) **row(Kiran, 1000)**

Kolkota ATM Kiran	T1	Juhu ATM Aamir	T2
R1(Bal)		R2(Bal)	
Bal = Bal +500		Bal = Bal - 1200	
W1(Bal)		W2(Bal)	
Abort		Commit	
EOT1		EOT2	

Schedule	Bal	
R1(Bal)	1000	
W1(Bal)	1500	Writes to stable storage
R2(Bal)	1500	
W2(Bal)	300	
Abort T1		For a consistent database state T2 should also be aborted
Commit T2		
EOT1		
EOT2		

Incorrect Summary Problem

A transaction reads partially updated data item values from other transactions

Account(Num,Name, Bal) row(1,Kiran,1000), row(2,Aamir,2000)

Kolkota ATM	Kiran	T1	Juhu	Aamir	T2
R1(2,Bal2)			R2(2,Bal2)		
Bal 2= Bal 2- 500			R2(1,Ba1)		
W1(2,Ball2)			Sum = (Bal1 + Bal2)		
R1(1,Bal1)			EOT2		
Bal1 = Bal1 + 500					
W1(1,Bal1)					
EOT1					

Schedule	Bal2	Bal1
R1(2,Bal2)	2000	1000
W1(2,Bal2)	1500	
R2(2,Bal2)	*1500*	
R2(1,Bal1)		*1000*
R1(1,Bal1)		1000
W1(1,Bal1)		1500
EOT1		

The correct sum calculated by T2 should be 3000

Why Recovery is Needed

When a transaction is submitted to a DBMS for execution, the system is responsible for making sure that either

- –all the operations in the transaction are completed successfully and their effect is recorded permanently in the database, or
- –the transaction has no effect whatsoever on the database or on any other transactions

The DBMS must not permit some operations of a transaction to be applied to the database while other operations of the transaction are not.

This may happen if a transaction fails after executing some of its operations but before executing all of them.

Types of Failures

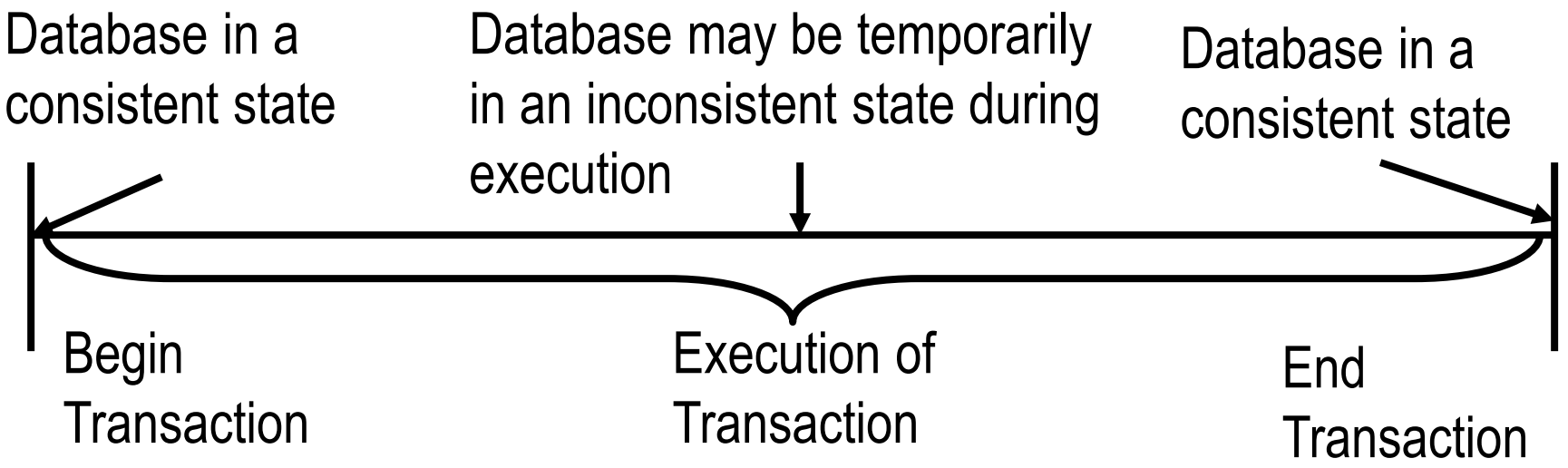
- A computer failure (system crash)
- A transaction or system error
- Local errors or exception conditions detected by the transaction
- Concurrency control enforcement
- Disk failure
- Physical problems and catastrophes

Types 1-4 are more common types of failures, whenever a failure of type 1 through 4 occurs, the system must keep sufficient information to recover from the failure.

Transaction Definition

A transaction is a unit of consistent and reliable computation

A database state consists of a set of values of all data items in the database. A database state is consistent if the database obeys all the integrity constraint



Transaction Definition

If the database was consistent before the transaction was executed, then it will be consistent after the transaction is executed, regardless of the facts that:

- transaction was executed concurrently with other transactions
- failures may have occurred during its execution

Transaction Operations

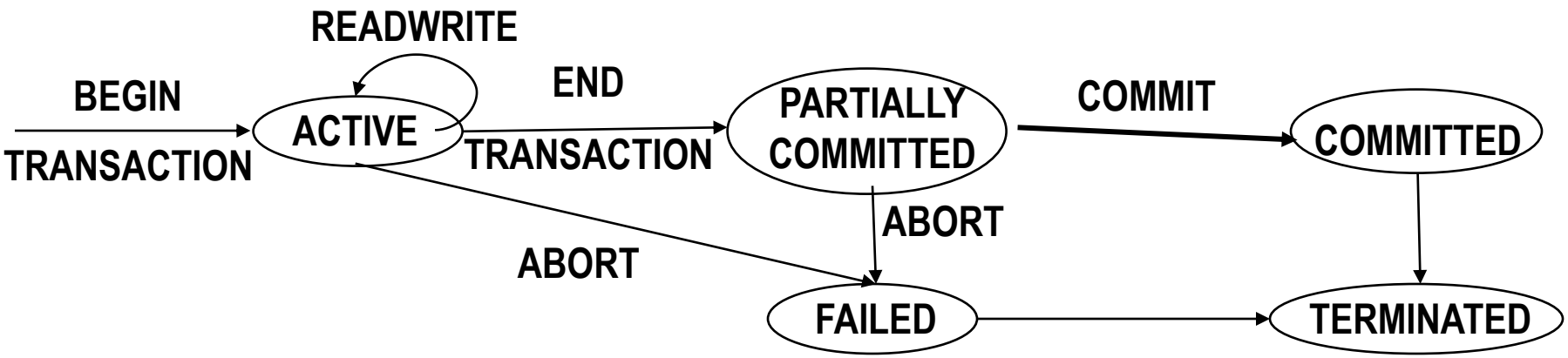
The system needs to keep track of transaction as it executes. This done by using following operations.

- BEGIN_TRANSACTION
- READ or WRITE (R(X) or W(X); read_item(X) or write_item(X))
- END_TRANSACTION
- COMMIT_TRANSACTION
- ROLLBACK OR ABORT

Some recovery techniques need additional operations:

- UNDO
- REDO

Transaction States



A Transaction goes into an active state immediately after it starts execution and issues Reads and Writes

When a transaction ends it moves to partially committed state

If the concurrency control algorithm and recovery technique imposed checks are successful the transaction enters committed state.

The transaction can enter abort state if the above checks are unsuccessful or from active state.

Terminated state corresponds to transaction leaving the system

Properties of Transactions

Atomicity

all or nothing;

if a transaction is interrupted due to a failure then all its actions must be undone.

Atomicity is preserved by using transaction recovery in case of transaction aborts; and crash recovery in case of system crashes

Consistency

no violation of integrity constraints;

a transaction which executes alone against a consistent database leaves it in a consistent state.

Properties of Transaction (Cont.)

Isolation

concurrent changes are invisible;

if several transactions execute concurrently, the results must be same as if they were executed serially;

incomplete transactions cannot reveal its results to other transactions before commitment

Durability

committed updates persist;

once a transaction commits, the system must guarantee that the results of its operations will never be lost (database recovery)

Serializability Theory & Concurrency Control

Basis for Transaction Processing

Since we are allowing multiple transactions to execute concurrently (interleaved fashion), we need to provide a guarantee that the properties of the transactions are maintained.

Serializability theory, concept of schedules forms the basis for evaluating the transaction processing without considering a particular DBMS.

Further, serializability theory enables us to define the concepts of recovery and concurrency control

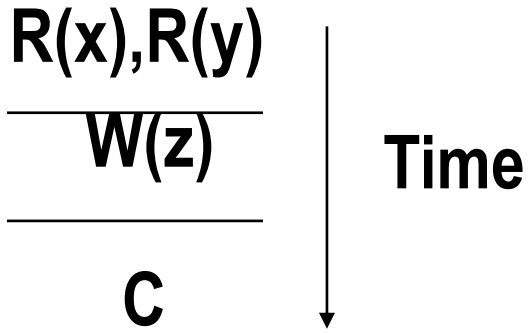
Finally, the serializability theory enables testing of new concurrency control algorithms

Serializability Theory

We evaluate the operations of a transaction that can be done in parallel or concurrently.

Example: T={R(x), R(y), z = x+y, W(z), C}

It is quite possible that R(x) and R(y) are done in parallel (or concurrently) followed by W(z). If so, how do we represent this kind of execution.



Serializability Theory

That is, we do not need to specify the order in which all the operations need to be executed, but only some.

This specification of order between transaction operations is done by "precedence relation" denoted by ``<``.

For transaction T we need not specify order of execution between R(x) and R(y).

Serializability Theory (Cont.)

Note

1. For every transaction T, we have a set of operations executed by the transaction, like R(x), W(x), C, A.
2. For each transaction these operations are sent to the data processor in some order by the scheduler.
3. This “order” in which these operations are sent to the data processor forms the basis for serializability theory.

Serializability Theory (Cont.)

This is done by using a partial order relationship ``<`` for each transaction T. Since all reads can be done concurrently (even on the same data item). No precedence relationship needs to be defined between reads.

But if there is a R(x) and W(x) then one of them must occur before the other. This is because the value read by R(x) depends on whether W(x) was executed before it or after it.

For Example:

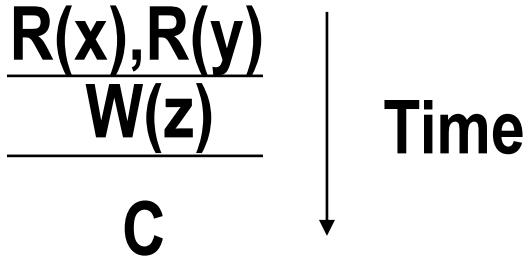
Let X = 10, consider R(X), X= X+ 10, W(X). R(X) reads X = 10.

Now, X = X+10, W(X), R(X). R(X) reads X = 20.

Ordering of Reads and Writes is very important.

Serializability Theory (Cont.)

Given that the operations of a single transaction T were done as follows



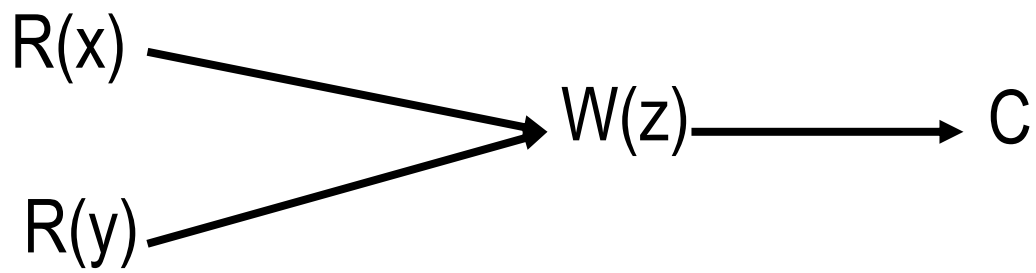
We can denote by S the set of operations issued by transaction T. Therefore, S = {R(x), R(y), W(z), C}.

Further, we can use < to denote the precedence relation between the operations.

Therefore, < = { (R(x), W(z)) , (R(y), W(z)) , (R(x), C) , (R(y), C) , (W(z), C) }

Serializability Theory (Cont.)

This can be shown by following DAG representation



DAG Representation of a Transaction T

Serializability Theory (Cont.)

By using this formal notation, we can in general, formally represent the operations performed by a transaction T_i as follows:

Let O_{ij} => operation $O_j \in \{R,W\}$ being performed by

transaction T_i on data item x

$OS_i = \cup_j O_{ij}$; $N_i \in \{C,A\}$ is terminal condition of T_i

A transaction T_i is a partial order $\{S_i, <_i\}$ such that

1. $S_i = OS_i \cup N_i$
2. For any two operations O_{ij}, $O_{ik} \in OS_i$, if $O_{ij} = R(x)$ and $O_{ik} = W(x)$, then either $O_{ij} <_i O_{ik}$ or $O_{ik} <_i O_{ij}$
3. $\forall O_{ij} \in OS_i$, $O_{ij} <_i N_i$.

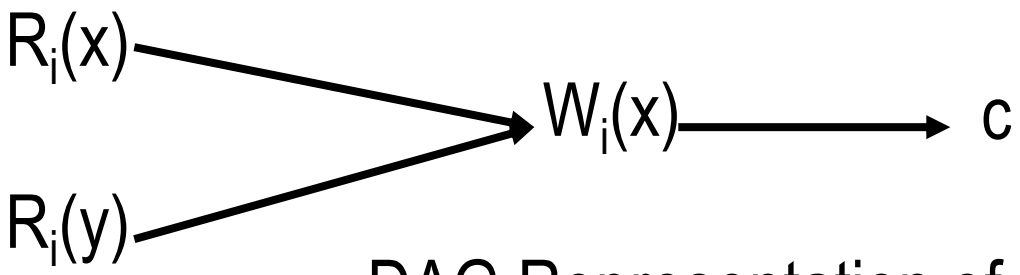
Serializability Theory (Cont.)

Example:

$T_i = \{R_i(x), R_i(y), x = x+y, W_i(x), C\}$

$S_i = \{R_i(x), R_i(y), W_i(x), C\};$

$<_i = \{(R_i(x), W_i(x)), (R_i(y), W_i(x)), (W_i(x), C), (R_i(x), C), (R_i(y), C)\}$



DAG Representation of a Transaction

Serializability Theory (Cont.)

What has been done till now?

Basic concurrency between simultaneous reads, and operations on different data items has been incorporated.

Reads and Writes on same data items performed by a transaction have been ordered by precedence relation.

Serializability Theory (Cont.)

Can we define precedence relationship when more than one transaction is considered?

Yes!!!

Let T_1, T_2,.... T_n be *n* transactions.

Let OS_1, OS_2,... OS_n be the sets of operations performed by the

above transactions, respectively.

Let $<_1$, $<_2$,..., $<_n$ be the precedence relationships with in each transaction, respectively.

Let S_1, S_2,..., S_n be the operations OS_i + {C,A} for each transaction T_i.

Serializability Theory (Cont.)

Define

1. $S_T = \cup_i S_i$
2. $<_T = \cup_i <_i$

What does this mean?

1) => we consider all the operations (including commits and aborts) of all the transactions together

2) => we maintain (or respect) the ordering of reads and writes on same data items with each transaction T_i (which is given by $<_i$)

Serializability Theory (Cont.)

Notation note: the subscript T in S_T or $<_T$ denotes all the transactions $\{T_1, T_2, ..., T_n\}$ together (and not a new or different transaction)

Have we represented completely all the operations performed by all the transactions?

- We have incorporated all the information about each of the transactions
- But we have not incorporated the relationship between concurrent access to same data item by different transactions; therefor this needs to be included

Serializability Theory (Cont.)

In S_T two operations are said to be conflicting operations if they both operate on the same data item and at least one of them is a write.

Given a S_T we can have conflicting operations as

- $R_i(x), W_j(x) \in S_T$, or
- $W_i(x), W_j(x) \in S_T$

Serializability Theory (Cont.)

We now define a precedence relationship between conflicting operations among different transactions. That is, we order these operations in time.

Therefore, if O_{ij} and O_{kl} are conflicting operations then

either, $O_{ij} <_T O_{kl}$ or $O_{kl} <_T O_{ij}$ where $O_{ij} \in OS_i$, and $O_{kl} \in OS_k$; $i <> k$;

Note that, two operations are conflicting if they both access the same data item X, and **at least one of the two operations is a write operation**.

Serializability Theory (Cont.)

Example

T_1: $\{R_1(x), x = x+1, W_1(x), C_1\}$; T_2: $\{R_2(x), x = x+1, W_2(x), C_2\}$

$S_1 = \{R_1(x), W_1(x), C_1\}$; $S_2 = \{R_2(x), W_2(x), C_2\}$

$S_T = \{R_1(x), W_1(x), C_1, R_2(x), W_2(x), C_2\}$

$<_1 = \{(R_1(x), W_1(x)), (R_1(x), C_1), (W_1(x), C_1)\}$

$<_2 = \{(R_2(x), W_2(x)), (R_2(x), C_1), (W_2(x), C_1)\}$

Conflicting operations between T_1 and T_2:

$\{ \underline{(R_1(x), W_2(x))} , \underline{(R_2(x), W_1(x))} , \underline{(W_1(x), W_2(x))} \}$

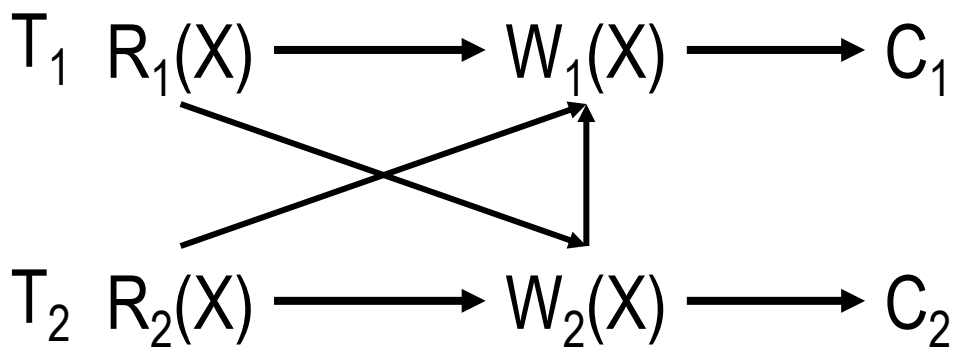
Initially, $<_T = <_1 \cup <_2$; i.e. no ordering of conflicting operations among different transactions.

Serializability Theory (Cont.)

Now for each pair of conflicting operations between transactions assign a precedence relation $\leq_T$, say,

1. $R_1(x)$, $W_2(x)$ assign $R_1(x) <_T W_2(x)$
2. $R_2(x)$, $W_1(x)$ assign $R_2(x) <_T W_1(x)$
3. $W_1(x)$, $W_2(x)$ assign $W_2(x) <_T W_1(x)$

$<_T$ now defines the complete schedule for the transactions T_1 and T_2



Complete Schedule

Complete Schedule

An interleaved order in which the individual operations of all the transactions are executed is known as a complete schedule

A complete schedule SC(T) over a set of transactions $T = \{T_1, T_2, ..., T_n\}$ is a partial order $SC(T)=\{S_T, <_T\}$ where

1. $S_T = \cup_i S_i$
2. $<_T = \cup_i <_i$
3. for any two conflicting operations O_{kl}, $O_{ij} \in S_T$, either $O_{ij} <_T O_{kl}$ or $O_{kl} <_T O_{ij}$

Complete Schedule

$T_1 = R_1(x) \rightarrow W_1(x) \rightarrow C_1$

$T_2 = R_2(x) \rightarrow W_2(y) \rightarrow W_2(x) \rightarrow C_2$

$T_3 = R_3(y) \rightarrow W_3(x) - W_3(y) \rightarrow W_3(z) \rightarrow C_3$

$R_2(x) \rightarrow W_2(y) \rightarrow W_2(x) \rightarrow C_2$

$H = R_3(y) \rightarrow W_3(x) \rightarrow W_3(y) \rightarrow W_3(z) \rightarrow C_3$

$R_1(x) \rightarrow W_1(x) \rightarrow C_1$

Precedence Graph

Precedence graph is a directed graph, with nodes/vertices as transactions, and edges denoting precedence relationship between conflicting operations of different transactions.

For a pair of conflicting operations $O_{ij} \in T_i$ and $O_{kl} \in T_k$,

- if $O_{ij} <_T O_{kl}$ then have an edge from T_i to T_k, <u>else</u>
- if $O_{kl} <_T O_{ij}$ then have an edge from T_k to T_i

(Represent multiple such edges by a single edge)

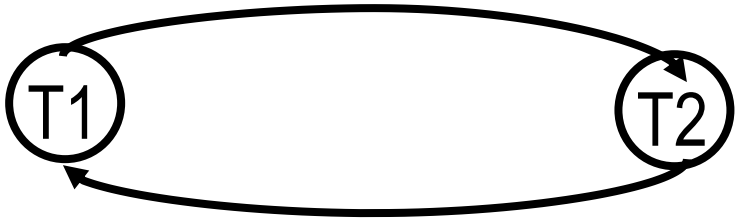
Precedence Graph

From our example, we have nodes corresponding to transactions T_1 and T_2

Edges: $R_1(X) <_T W_2(X)$ => edge T_1 -> T_2;

$R_2(X) <_T W_1(x)$ => edge T_2 -> T_1, and

$W_2(X) <_T W_1(X)$ => edge T_2 ->T_1



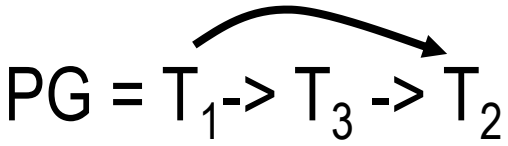
A cycle in Precedence Graph implies potential for database inconsistency

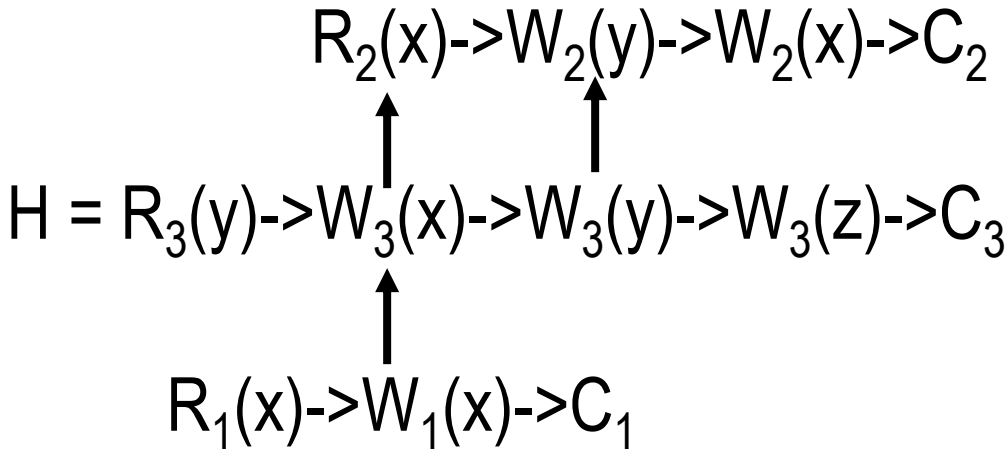
Precedence Graph (Cont.)

$T_1 = R_1(x) \rightarrow W_1(x) \rightarrow C_1$

$T_2 = R_2(x) \rightarrow W_2(y) \rightarrow W_2(x) \rightarrow C_2$

$T_3 = R_3(y) \rightarrow W_3(x) - W_3(y) \rightarrow W_3(z) \rightarrow C_3$





T1 -> T3 is in PG because $W_1(x) < W_3(x)$;

T1 -> T2 is in PG because $W_1(x) < W_2(x)$

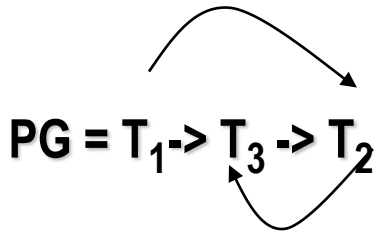
No cycle in precedence graph so no database inconsistency.

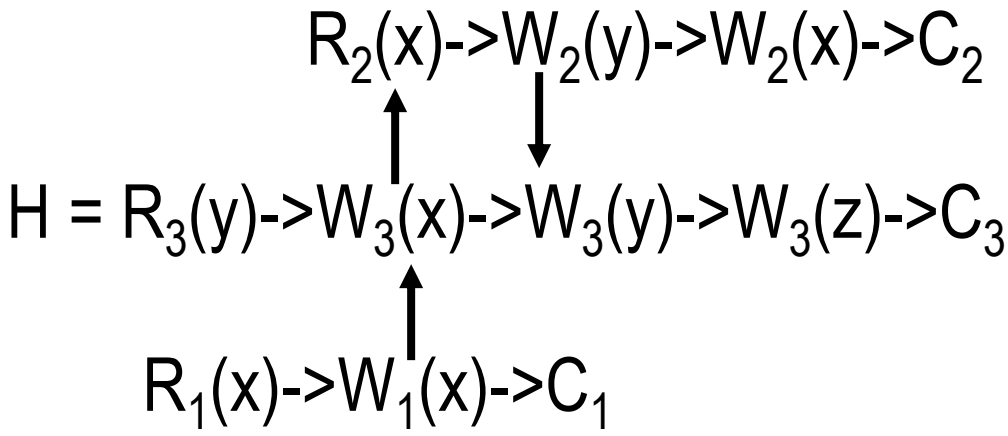
Precedence Graph (Cont.)

$T_1 = R_1(x) \rightarrow W_1(x) \rightarrow C_1$

$T_2 = R_2(x) \rightarrow W_2(y) \rightarrow W_2(x) \rightarrow C_2$

$T_3 = R_3(y) \rightarrow W_3(x) - W_3(y) \rightarrow W_3(z) \rightarrow C_3$





PG has a cycle; the database may not be consistent.

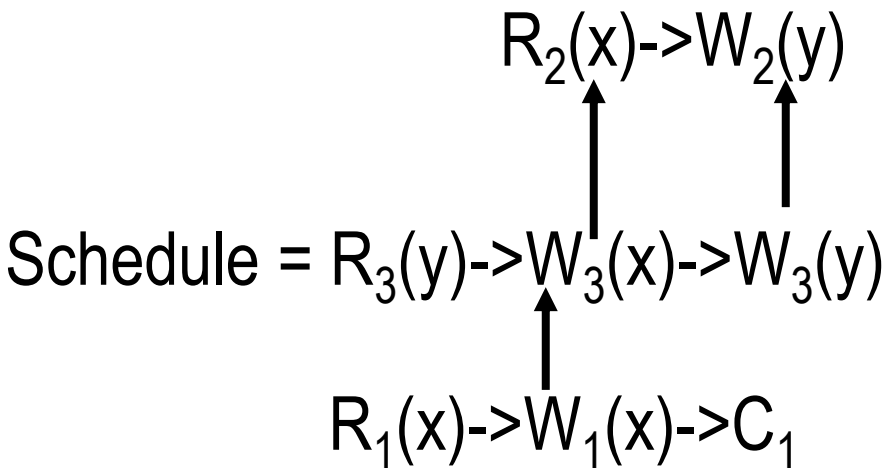
Schedule

A schedule is a prefix of a complete schedule such that only some of the operations and only some of the ordering relationships are included

$T_1 = R_1(x) \rightarrow W_1(x) \rightarrow C_1$

$T_2 = R_2(x) \rightarrow W_2(y) \rightarrow W_2(x) \rightarrow C_2$

$T_3 = R_3(y) \rightarrow W_3(x) - W_3(y) \rightarrow W_3(z) \rightarrow C_3$



The scheduler keeps building the Precedence Graph as the operations of active transactions are scheduled so that the PG is acyclic.

The scheduler delays scheduling those operation that can cause a cycle in PG

Classification of Schedules

Serial Schedule

- all actions of a transaction occur consecutively;
- no interleaving of transactions

Serializable Schedule

- the net effect of a schedule upon a database is equivalent to some serial schedule;
- transactions execute concurrently PG is acyclic
- Also known as conflict serializable schedules

Recoverable Schedules

A schedule S is said to be recoverable if no transaction T in S commits until all transactions T' that have written an item that T reads have committed.

A transaction T is said to read from transaction T' in as schedule S if some item X is first written by T' and later read by T.

S_a: $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; c_2; $w_1(Y)$; c_1;

S_a is recoverable.

Recoverable Schedules

S_c: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; c_2; a_1

S_c is not recoverable because T_2 reads an item from T_1,

and then T_2 commits before T_1.

S_d: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$, c_1; c_2 is recoverable.

In a recoverable schedule no committed transaction ever needs to be rolled back.

Cascading Rollback

It is possible that an uncommitted transaction has to be rolled back because it read an data item from an transaction that aborted.

This gives rise to a problem of cascading rollback (or cascading abort).

S_e: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$, a_1

T_2 has to be rolled back because T_1 is aborted.

A schedule is said to avoid cascading rollback if every transaction in the schedule only reads items that were written by committed transactions.

Strict Schedule

A strict schedule is a schedule in which transactions can neither read nor write an item X until the last transaction that wrote X has committed or aborted.

S_f: $w_1(X=5)$, $w_2(X=8)$, a_1 (original value of X is 9)

If T_1 aborts, the system rolls back the value of X to 9, even though T_2 is successful (thus losing update $w_2(X=8)$; strict schedule would not have allowed $w_2(X)$ before T_1 commits or aborts (whereas a recoverable schedule would have allowed it - blind write).

Concurrency Control Algorithms

The primary function of a concurrency control algorithm is to generate a serializable schedule for the execution of pending transactions

- Two Phase locking (2PL)
- Timestamp Ordering (TO)

Locking Approach

Basic Concept

For each data item there is a lock attached;

Before a transaction T_1 is given an access to a data item the scheduler examines the associated lock.

If no transaction holds the lock the scheduler gets the lock for transaction T_1;

If there is some other transaction holding the lock; the scheduler waits till the lock is released.

Thus at a given time only one transaction accesses the data item (mutual exclusion).

Locking Approach

Since transactions access data items for a read access or a write access; there are two kinds of locks defined as

- rl(x) a read lock for data item x;
- wl(x) a write lock for data item x.

Compatibility of Locking Modes

A transaction T_i wanting to read a data item x obtains read lock $rl_i(x)$, to write a data item it obtains write lock $wl_i(x)$.

Two locking modes are compatible if two transactions which access the same data item can obtain these locks on that data item at the same time.

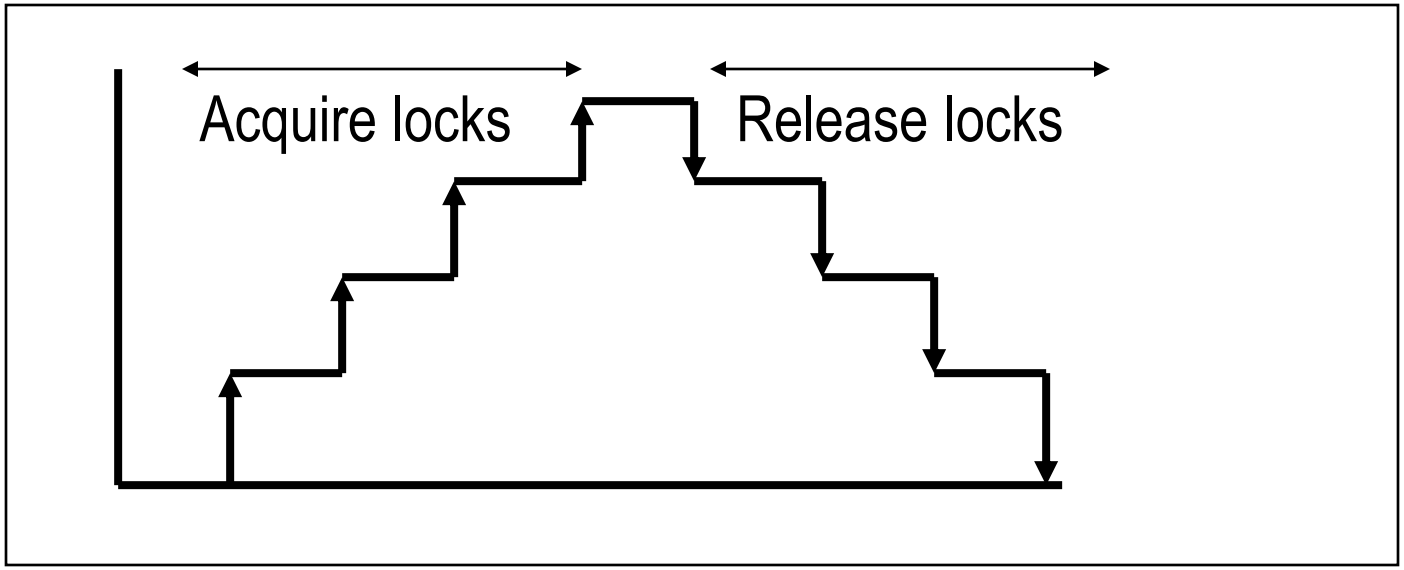
	$rl_i(x)$	$wl_i(x)$
$rl_j(x)$	compatible	not compatible
$wl_j(x)$	not compatible	not compatible

Locking facilitates concurrent processing of transactions

Locking concept is extensible so as to include additional modes of locking (for e.g., incr, decr)

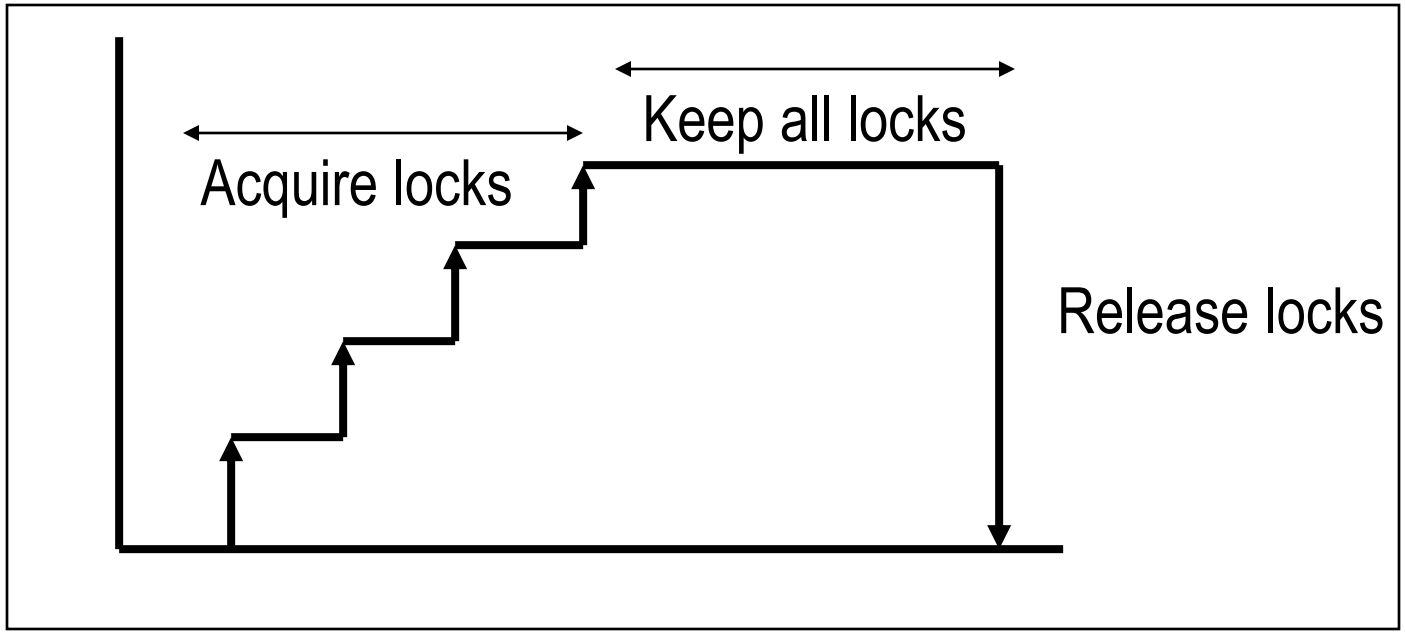
Two-Phase Locking

1. Transaction must obtain the lock before accessing the data item
2. When the data item is locked by another transaction, the transaction must wait till the lock is released
3. When a transaction releases a lock it should not request for any more locks



May give rise to cascading aborts (see, levels of isolation)

Strict Two Phase Locking



Locks are held until the end of the transaction.

No cascading aborts, but reduces concurrency.

Basic Timestamp Ordering

Each transaction T_i is issued a globally unique (monotonically increasing) timestamp $ts(T_i)$.

The transaction manager assigns timestamp of a transaction to each of the operations issued by the transaction.

Conflicting operations are resolved by the timestamp order.
Each data item x has two time stamps

- ts(w(x)) for write
- ts(r(x)) for read

Largest timestamp among all transactions performing the write/read operations.

Basic Timestamp Ordering

For read operation $r(x) \in T_i$

- if $ts(T_i) < ts(w(x))$ restart T_i;
- else process r(x) – update ts(r(x))as the Largest timestamp among all transactions performing the r(X)

For write operation $w(x) \in T_i$

- if $ts(T_i) < ts(w(x))$ or $ts(r(x))$ restart T_i;
- else process w(x) – update ts(w(x)) as the Largest timestamp among all transactions performing the w(X)

These rules are checked before scheduling each read/write operation.

Guarantees conflict serializability. Can cause too many transaction restarts.

Recovery

Failures in DBMSs

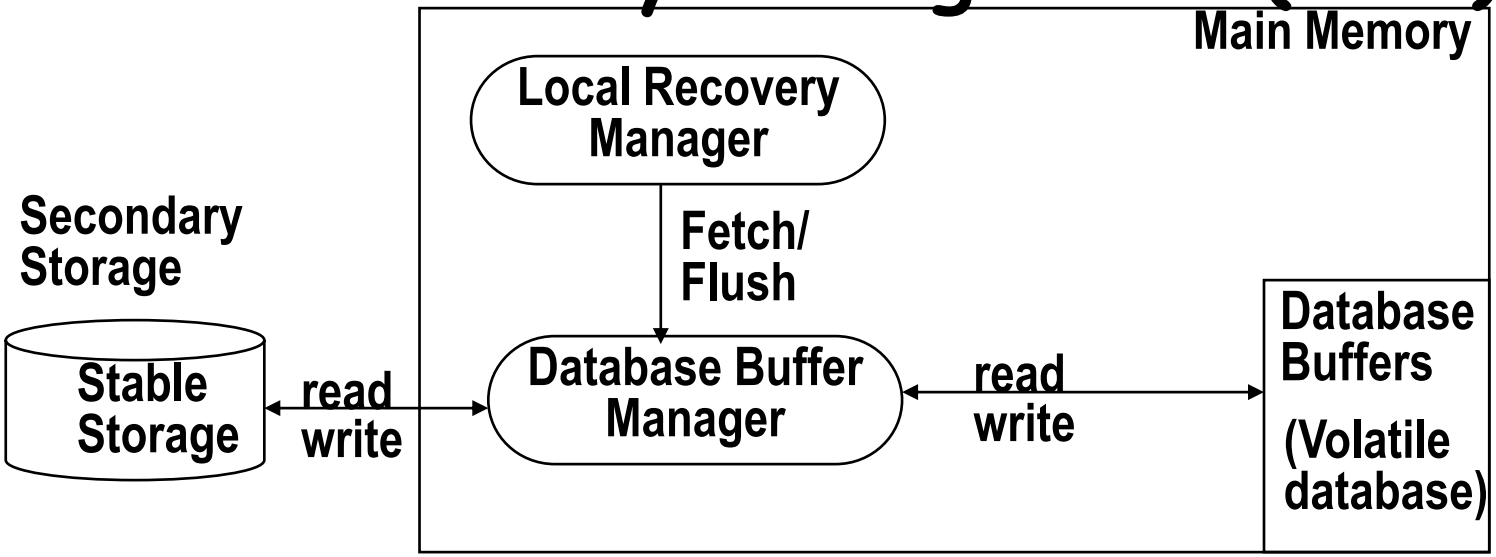
Transaction Failures

- Transaction aborts due to software bugs, or deadlocks
- Average of 3% of transactions abort abnormally (not user-intended)

Media Failures

- Failure of the secondary storage devices

Local Recovery Management (LRM)



Fetch: read a page from stable storage to database buffer.

Flush: force a write of a page from database buffer onto the stable storage.

Local database systems under system failures result in losing of the volatile database.

In order to recover from system failures the DBMS has to maintain some additional information to facilitate recovery.

Recovery Information

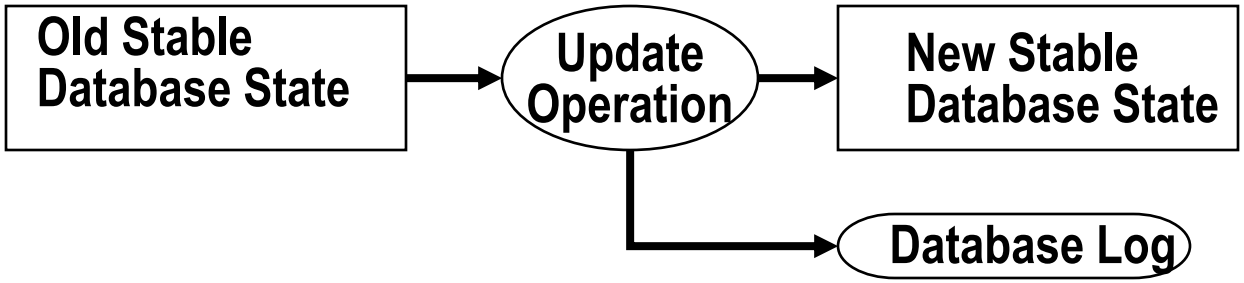
In-place Updating

Physically changes the values of the database items.

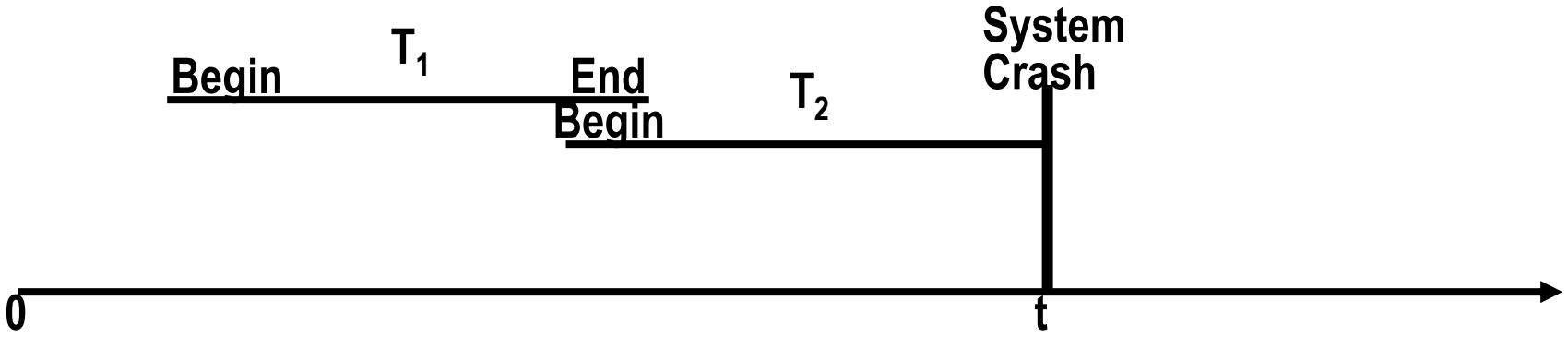
In-place updates cause previous values of the affected data items to be lost. Therefore, enough information needs to be kept about database state changes to facilitate recovery of the database to a consistent state.

This information is typically maintained in database log.

In-Place Update Recovery



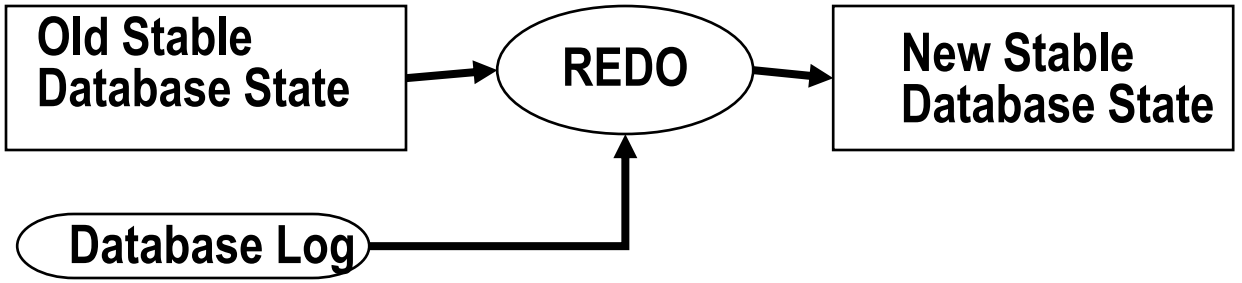
For every update, the DBMS must not only perform the update but also must write a log record to an append only file



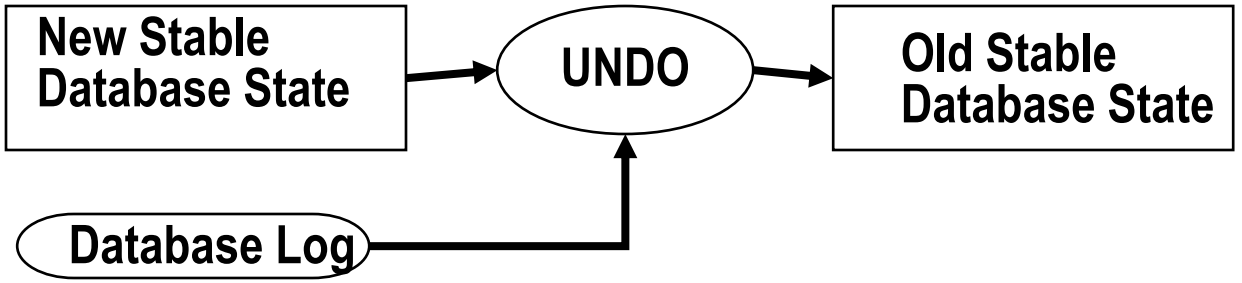
Assume that buffer page updates are reflected in stable storage only when a page is replaced. Upon recovery the stable database must reflect updates done by T_1 and not those done by T_2

In-Place Update Recovery

Redo T_1's updates:

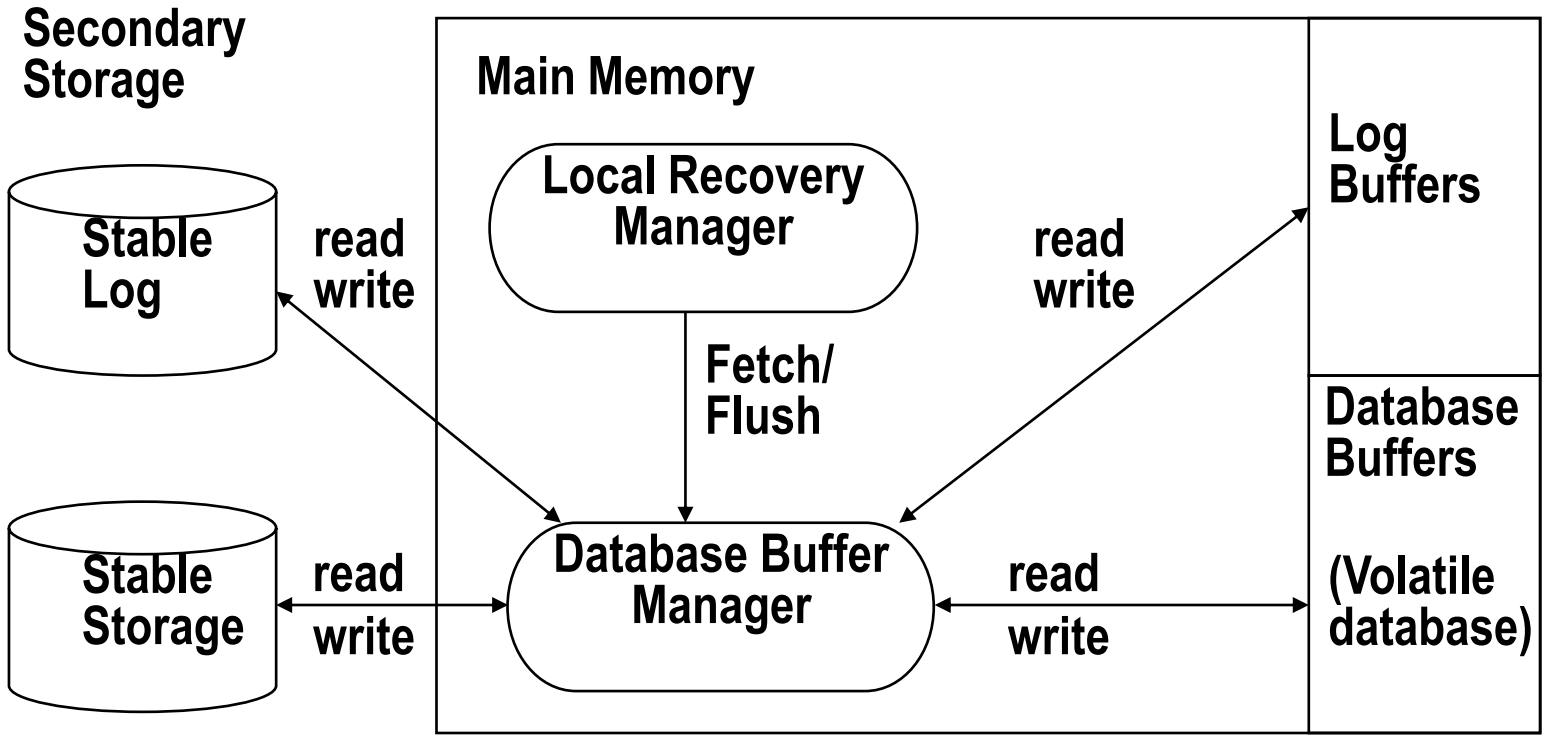


Undo T_2's updates:



The log contains information like: transaction-id, begin transaction record, before update image, after update image.

Logging



Log is also maintained in the main memory buffers (log buffers).

Logging can be reflected in the stable storage synchronously (force after each write) or asynchronously (leave it to buffer manager).

Logging

Write-ahead logging protocol

- For undo: before the stable storage is updated, the before images must be stored in stable log.

- For redo: before transaction commits, the after images must be stored in stable log prior to the updating of the stable database.

Logging Actions

For each of the transactions commands following logging actions need to be taken.

Begin Transaction

LRM writes begin transaction record into the log.

Read

If the data item is in the buffer, LRM reads from the buffer; else issues a fetch to the buffer manager to make the data available from the stable storage.

Write

If the data item is available in the buffer its value is modified; else it is fetched from stable storage; the before and after images of the data item values are recorded in the log.

Dependence between LRM and Buffer Manger

Fix (Deferred Update)/no-fix decision (no-steal/steal; pin/no-pin):

Whether the buffer manager writes buffer pages updated by transactions into stable storage during the execution of the transaction, or has to wait for the LRM to instruct it write them back.

That is, once a LRM issues “fix” on a page in database buffer it cannot be replaced by the database buffer manager. An explicit “unfix” command allows the page to be free to be replaced by database buffer manager.

This is useful, when an active transaction updates a data item in database buffer. By fixing the page with the data item, the page does not get written on stable storage, and in case of system crash the stable database does not have the affect of updates by active but not committed transactions on recovery and no undo is required.

Dependence between LRM and Buffer Manger

Flush/no-flush decision:

Whether the buffer manager is forced flush the buffer pages updated by a transaction into the stable storage at the end of that transaction, or the buffer manager flushes them out whenever it needs to according to its buffer management algorithm.

This is useful, as before transaction commit, the LRM issues a flush, and all the updates made by transaction are reflected in the stable storage. Hence on recovery after system crash, the affect of all committed transactions is there on the stable database and no redo is required.

Dependence between LRM and Buffer Manger

Four execution strategies:

- no-fix/no-flush (Undo/Redo)
- no-fix/flush (Undo/no-Redo)
- fix/no-flush (no-Undo/Redo)
- fix/flush (no-Undo/no-Redo)

Based on these execution strategies the LRM on recovery has to perform Redo & Undo operations

Checkpoints

To reduce the amount of redo and undo operations check pointing is used.

A checkpoint is the latest database state in time which is consistent

All undo and redo operations can be done from checkpointed database state onwards.

Steps

1. Write begin checkpoint (no new transactions accepted).
2. Complete all the active transactions flush all the updated pages.
3. Write end checkpoint record.

Handling Media Failures