Transaction Management

Dr G.N.Wikramanayake

Dr Jeevani Goonetillake

University of Colombo School of Computing

Concurrency Control

- Concurrency control deals with influencing how data can be viewed and updated by users accessing the same information at one time.
- Concurrency control allows users to use the database concurrently without damaging the transactions of other users.
- It supports and ensures the availability and correct operations of simultaneous multiple access in the database system.

Concurrency Control

- **Single user** – at most one user at a time can use the system. Restricted to some PC DBMS.

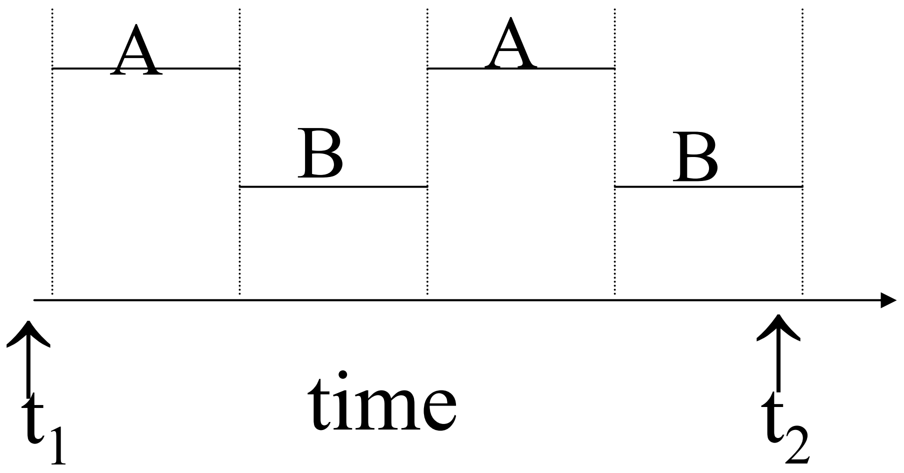
- **Multi-user** – many users can use the system concurrently (at the same time). Most DBMS are multi-user. Airline reservations systems, banks, insurance agencies, stock exchanges are multi-user systems operated concurrently.

Multiprogramming

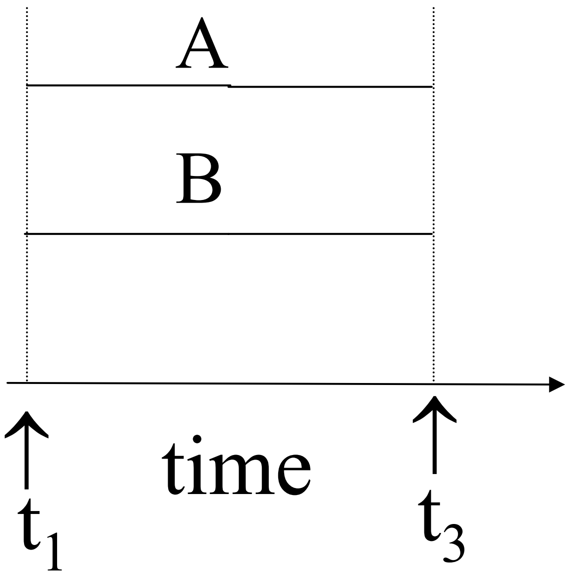
Multiple users can use computer systems simultaneously because of the concept of **multiprogramming**. When only one CPU, the multiprogramming operating systems execute some commands from one program, then suspend that program and execute some commands from the next program and so on. A program is resumed at the point where it was suspended when it gets its turn to use the CPU again. Hence, concurrent execution of the program is actually **interleaved**. Simultaneous processing of multiple programs are done with multiple CPUs.

Interleaved model of concurrent execution

Single CPU



Multiple CPUs



Database Access Operations

The basic unit of data transfer from the disk to the computer memory is one block. For discussion purpose, consider transactions at the level of data item (field of some record in the database) and disk blocks. At this level the database access operations that a transaction can include are

- **READ(X)** - reads database item X into a program variable X;
- **WRITE(X)** - write the value of program variable X into the database item X.

- **Executing a READ(X)**

```
SELECT labmark INTO old_mark FROM enrol
WHERE studno = sno and courseno = cno
FOR UPDATE OF labmark;
```

  1. Find the address of the disk block that contains item X.
  2. Copy that disk block into a buffer in main memory (if not already in some main memory buffer).
  3. Copy item X from the buffer to the program variable named X.

- **Executing a WRITE(X)**

```
UPDATE enrol SET labmark = new_mark
WHERE studno = sno and courseno = cno;
```

  1. Find the address of the disk block that contains item X.
  2. Copy that disk block into a buffer in main memory (if not already in some main memory buffer).
  3. Copy item X from the program variable named X into its correct location in the buffer.
  4. Store the updated block from the buffer back to disk (either immediately or at some later point of time)

Transaction States and additional operations

- A transaction is an atomic unit of work that is either completed in its entirety or not done at all. For recovery purpose, the system needs to keep track of when the transaction starts, terminates and commits or aborts. The recovery manager keeps track of:

BEGIN marks the beginning of transaction execution

READ or WRITE read or write operations on the database items that are executed.

END specifies that READ and WRITE transaction operations have ended and mark the end of transaction execution.

Transaction States and additional operations

COMMIT signals a successful end of the transaction so that any changes (updates) executed by the transaction can be safely committed to the database and will not be undone.

ROLLBACK (or ABORT) signals that the transaction has ended unsuccessfully so that any changes or effects that the transaction may have applied to the database must be undone.

Properties of Transactions

Atomicity - A transaction is an atomic unit of processing. It 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 failures.

Transaction Properties …

e.g. Transfer 50 from account A (A=1000) to B (B= 2000)

```
T1: BEGIN
      READ(A);
      A = A – 50;
      WRITE(A);
      READ(B);
      B = B + 50;
      WRITE(B);
    END;
```

A=950; B=2050;

Consistency - take the database from one consistent state to another

Value of A+B (3000) should be same before transaction and after transaction

Atomicity - either performed in its entirety or not performed at all

Transaction failure after WRITE(A), but before WRITE(B), then A=950; B=2000; i.e. 50 is lost

Data is now inconsistent as A+B is now 2950

Durability - changes must never be lost because of subsequent failures

Recover database: remove changes of a partially done transaction (A=1000; B=2000); reconstruct completed transactions (A=950; B=2050)

Isolation - updates not visible to other transactions until committed

Between WRITE(A) and WRITE(B) if second transaction reads A and B it sees inconsistent data as A+B = 2950

Problems with Concurrent Use

E.g. **Transaction T1** - No of reservations for airline A is X; No of reservation for airline B is Y; N reservation from A is cancelled and booked for B.

Transaction T2 - M reservations to airline A.

T1	**T2**
READ(X)	READ(X)
X = X – N	X = X + M
WRITE(X)	WRITE(X)
READ(Y)	
Y = Y + N	
WRITE(Y)	

Several problems can occur when concurrent transactions execute in an uncontrolled manner.

1. The lost update problem

This occurs when two transactions that access the same database item have their operations interleaved in a way that makes the value of some database item incorrect.
X=80; Y=100

T1	T2	
READ(X)		X = 80, N = 5, M = 4
X = X – N		X = 75
	READ(X)	X = 80
	X = X + M	X = 84
WRITE(X)		X = 75
READ(Y)		
	WRITE(X)	X = 84
Y = Y + N		Y = 105
WRITE(Y)		T1: X+Y = 84+105=189

but X should be 80-5+4 = 79

2. The temporary update (Dirty read) problem

This occurs when one transaction updates a database item and then the transaction fails for some reason.

T1	T2	
READ(X)		X = 80, N = 5, M = 4
X = X – N		X = 75
WRITE(X)		X = 75
	READ(X)	X = 75
	X = X + M	X = 79
	WRITE(X)	X = 79
READ(Y)		
ROLLBACK		

- abort - changes X back to its original value gives X = 80

but should be 80+4 = 84

3. The incorrect summary problem

If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and

T1	T2	
	sum = 0	
READ(X)		
X = X – N		
WRITE(X)		
	READ(X)	
	sum = sum + X	
	READ(Y)	
	sum = sum + Y	sum=X+Y=75+100=175
READ(Y)		
Y = Y + M		
WRITE(Y)		

4. Unrepeatable Read problem

Another problem that may occur is the **unrepeatable read** where a transaction T2 reads an item twice (i.e. X) and the item is changed by another transaction (i.e. T1) between the two reads.

T1	**T2**	
	…..	
	READ(X)	X=80
READ(X)		
X = X – N	…..	
WRITE(X)		
	READ(X)	X=75
	….	

Phantom Phenomenon

Set of rows that is read once might be different due to insert of new record.

T1	T2
	…..
	SELECT X 3 records
INSERT(X)	
	…..
	SELECT X 4 records
	….

Concurrency Control

- Concurrency control deals with influencing how data can be viewed and updated by users accessing the same information simultaneously.

  Do you want one user to view/change an order that is being changed/viewed by another user?

- There are two classes of concurrency control:

  (i) applies to read-only database access;

  levels of isolation: dirty read, committed read, repeatable read

  (ii) applies to updating database records: serializable

Dirty Reads

- Database server process reads from the database table without checking for locks (let this process look at dirty data). This can be useful when the table is static; 100% accuracy is not as important as speed and freedom from contention; you cannot wait for locks to be released.

SQL Syntax:

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITED;

All are possible {dirty read, non-repeatable, phantom}

Committed Reads

- Database server process reads rows from the database after seeing that lock could be acquired (do not let this process look at dirty data). This can be useful for lookups; queries; reports yielding general information (e.g. month-ending sales analyses).

SQL Syntax:

```
SET TRANSACTION ISOLATION LEVEL
READ COMMITTED;
```

Dirty read not possible; non-repeatable & phantom possible

Repeatable Reads

- Database server process puts locks on all rows examined to satisfy the query (do not let other processes change any of the rows I have looked at until I am done). It can be used for critical, aggregate arithmetic (e.g. account balancing); coordinated lookups from several tables (e.g. reservation systems).

SQL Syntax:

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

Dirty read and non-repeatable not possible; phantom possible

Serializable

- Always guarantee correct execution of transaction.

SQL Syntax:

SET ISOLATION TO SERIALZABLE;

All are not possible {Dirty read, non-repeatable, phantom}

Locking

- Concurrency control is enforced using locking: database level; table level; page level; row level; key level
- **Database Level Locking**: Other users cannot access database. Database stores exclusive. It can be used when executing a large number of updates involving many tables; archiving the database files for backups; altering the structure of the database.

Table Level Locking

- Other users cannot modify the table. It can be used to: avoid conflict with other users during batch operations that affects most or all of the rows of a table; avoid running out of locks when running an operation as a transaction; prevent users from updating a table for a period of time; prevent access to a table while altering its structure or creating indexes.

Table Level Locking

- **Table Level Locking in Share Mode**: Others may SELECT from the table.

SQL Syntax:

LOCK TABLE table_name IN SHARE MODE

- **Table Level Locking in Exclusive Mode**: Others may not SELECT from the table.

SQL Syntax:

LOCK TABLE table_name IN EXCLUSIVE MODE

Lock/Unlock

- **Unlocking a Table**:

SQL Syntax: UNLOCK TABLE table_name

- **Setting the Lock Mode**:
- Wait forever for the lock to be released.

SQL Syntax: SET LOCK MODE TO WAIT

- Do not wait for lock to be released.

SQL Syntax: SET LOCK MODE TO NOT WAIT

- Wait 20 seconds for lock to be released.

SQL Syntax: SET LOCK MODE TO WAIT 20

Serializability of Schedules

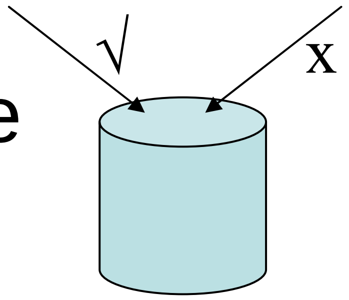
Schedule (History) - A schedule S of n transactions T1, T2, …, Tn is an order of the operations of the transactions subject to the constraint that operation of Ti in S must appear in the same order in which they occur in Ti.

Serial Schedules - For every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule. Otherwise the schedule is called non-serial. Serial schedules are always correct.

Serializable - If two disjoint groups of the non-serial schedules are equivalent to one of the serial schedules. Otherwise non-serializable.

Guaranteeing Serializability

- Protocols or set of rules are used to guarantee serializability.
- **locking** data items to prevent multiple transactions from accessing the item concurrently.
- **timestamps**, where a unique identifier for each transaction generated by the system. [immediate update]
- **multi-version**, where multiple versions of a data item is used. [shadow paging]



Locking Techniques

- Two types of locks:
  - Binary – can have two states or values, Locked and unlocked;
  - Shared and Exclusive locks – read_locked item is called shared locked; write_locked item is called exclusive locked.

Two-phase locking

- **Guaranteeing Serializability by Two-phase locking**
- If all locking operations precede the first unlock operation in the transaction such a transaction can be divided into 2 phases
- Expanding or growing phase, where new locks on items can be acquired but none can be released and Shrinking phase, where existing locks can be released but no new locks can be acquired.

Two-phase locking

- If every transaction in a schedule follows the two-phase locking protocol the schedule is guaranteed to be serializable, eliminating the need to test for serializability of schedules any more.
- Locking can be used to solve the concurrency control problems, but it can also lead to the problem of deadlock.
- Deadlock - occurs when each of two or more transactions are in a simultaneous wait state, each of them waiting for others to release a lock before it can proceed.

Deadlock

T1	T2	
read_lock(Y)		
READ(Y)		
	read_lock(X)	
	READ(X)	
…..	…..	
write_lock(X)		wait
	write_lock(Y)	wait

Deadlock Handling

- Two main methods for dealing with the deadlock problem: deadlock prevention and deadlock detection & recovery.

Deadlock Prevention method

- Uses deadlock prevention protocol to ensure that the system will never enter a deadlock state.
  - Each transaction locks all its data before it begins execution.
  - Either all requested data items are locked in one step or none are locked.

Deadlock Prevention

Disadvantages:

- low data utilisation: many data items may be locked but unused for a long period of time
- possible starvation: a transaction which requires a number of data items for its operation may find itself in a indefinite wait state while at least one of the data items is always locked by some other transaction.

Deadlock Detection

Allows the system to enter a deadlock state, but examines the state of the system periodically to detect whether a deadlock has occurred.

If it has, the system attempts to recover from the deadlock.

Deadlock Detection Process

- Keep information about the current locks of data items to different transactions, as well as any outstanding locking request for data items.
- Invoke an algorithm which uses this information to determine whether the system has entered a deadlock state. A typical technique is to use the Wait-for-Graph (WFG) and periodically invoke an algorithm to search for cycles in the graph. Each transaction involved in the cycle is said to be deadlocked.

Recovery Aspects

- The most common solution is rollback one or more transactions so that the deadlock can be broken.
- Three issues are involved in deadlock recovery
  - issue of choosing a victim - determine which transaction(s) among a set of deadlocked transactions to rollback to break the deadlock.
  - Issue of rollback operation - determine how far the chosen victim transaction should be rolled backed (total or partial).
  - Issue of starvation - avoid a situation where some transaction may always be chosen as the victim due to selections based on cost factors. This may prevent the transaction from ever completing its job.

Comparison

- Both methods may result in transaction rollback
- both methods require overheads
- prevention method is commonly used if the probability of the system entering a deadlock state is relatively high
- Otherwise detection and recovery method should be used

Checking for Serializability

- Consider the following two schedules. If they are executed as two serial schedules T1, T2 or T2, T1 then serializability is guarantee.

T1	**T2**
read_lock(Y)	read_lock(X)
READ(Y)	READ(X)
unlock(Y)	unlock(X)
write_lock(X)	write_lock(Y)
READ(X)	READ(Y)
X = X + Y	Y =X + Y
WRITE(X)	WRITE(Y)
unlock(X)	unlock(Y)

If initial values X=20, Y=30 then

T1, T2 would give X=50, Y=80

T2, T1 would give X=70, Y=50.

Checking for Serializability

- Assuming that there are no techniques used to guarantee serializability (e.g. two-phase locking is nor used) If T1, T2 are executed concurrently the schedule will be serializable only if it gives the result one of the above two serial schedules.

E.g., the following schedule is non-serializable.

T1

read_lock(Y)

READ(Y)

unlock(Y)

T2

read_lock(X)

READ(X)

unlock(X)

write_lock(Y)

READ(Y)

Y =X + Y

WRITE(Y)

unlock(Y)

T1

write_lock(X)

READ(X)

X = X + Y

WRITE(X)

unlock(X)

would give

X=50, Y=50

Timestamp Ordering

Another method for determining the serializability. There is no deadlock and no locks.

Basic idea is if a transaction A starts before transaction B then A should behave as if it completed entirety before B started – i.e. as a serial schedule.

Transaction A is assigned a unique timestamp TS(A) before starting executing the transaction

Next Transaction B is assigned TS(B) where TS(A) < TS(B)

WRITE-TS(X) denotes the largest timestamp of any transaction that executed WRITE(X) successfully

READ-TS(X) denotes the largest timestamp of any transaction that executed READ(X) successfully

Timestamp Ordering Protocol

Suppose transaction A issues READ(X)

- If TS(A) < WRITE-TS(X), then A needs to read a value of X that was overwritten by another transaction say B [A should never be allowed to see B's updates]. Hence Rollback A.
- If TS(A) ≥ WRITE-TS(X), then READ(X) is executed and READ-TS(X) = MAX{TS(A), READ-TS(X)}

Suppose transaction A issues WRITE(X)

- If TS(A) < READ-TS(X), then value of X that A is producing was needed previously, and system assumed that it would never change [A should never be allowed to update anything that B has already seen]. Hence Rollback A.
- If TS(A) < WRITE-TS(X), then attempting to write an obsolete value of X [A should never be allowed to update anything that B has already change]. Hence Rollback A.
- Otherwise WRITE(X) is executed and WRITE-TS(X) = MAX{TS(A), WRITE-TS(X)}

Timestamp Ordering Protocol

TS(T1)=1 TS(T2)=2

T1	READ-TS(X)	WRITE-TS(X)	T2	READ-TS(Y)	WRITE-TS(Y)
	0	0	**T2**	0	0
T1					
READ(Y)				1	0
			READ(Y)	2	0
			Y = Y – 500		
			WRITE(Y)	2	2
	1	0			
READ(X)	2	0	READ(X)		
Z = X + Y			X = X + 500		
	2	2	WRITE (X)		

Both T1 and T2 are successfully completed. Similar to T1, T2

Timestamp Ordering Protocol

TS(T1)=1 TS(T2)=2

	READ-TS(X)	WRITE-TS(X)		READ-TS(Y)	WRITE-TS(Y)
T1	0	0	**T2**	0	0
READ(X)	1	0			
			READ(Y)	2	0
			Y = Y – 500		
			WRITE(Y)	2	2
READ(Y)	1<2 Rollback ⇦				
Z = X + Y					
			READ(X)		
			X = X + 500		
			WRITE (X)		

T1 Rollback

Timestamp Ordering Protocol

There are schedules that are possible under timestamp but not possible under two-phase locking

There are schedules that are possible under two-phase locking but not possible under timestamp

Recovery from Failure

- Three types of failures: transaction, system and media failure. Recovery allows a database system to recover from physical or software failures when they occur in the system.
- If a transaction fails after executing some of its operations but before executing all of them. System failure, also called soft crash.

  The volatile storage is destroyed (e.g. power failure). This affects all transactions currently in progress but do not cause damage to the database.

Types of Failures

1. A computer failure (system crash)
   - A hardware or software error occurs in the computer system during transaction execution.

     E.g. Hardware error, internal memory lost.
2. A transaction or system error
   - Some operation in the transaction may cause the failure. E.g. integer overflow, division by zero, erroneous parameter values, logical programming error. User may interrupt using control-C.

Recovery from Failure

3. Local errors or exception conditions detected by the transaction.
   - During transaction execution, certain conditions may occur tat necessitate cancellation of the transaction. Done using programmed ABORT. E.g. data value not found, insufficient account balance.
4. Concurrency control enforcement.
   - Concurrency control method may decide to abort the transaction (e.g. violates serializability) or to be restarted later (e.g. several transactions are in a state of deadlock).

Recovery from Failure

5. Disk failure
   - Some disk blocks may lose their data (a read or write malfunction a disk read/write head crash) while reading or writing a transaction.

6. Physical problems and disasters
   - Power or air-conditioning failure, fire, theft, sabotages, overwriting disks or tapes by mistake, mounting of a wrong tape.

Failure types 1-4 occur more commonly than the types 5-6.

Recovery via Reprocessing

- Go back to a known point and reprocess the workload – periodically make copies of the database (save).
- Keep a record of all transactions since the copy.
- When failure occurs restore the database from the save and reprocess all transactions.
- This strategy is often infeasible, as same amount of time is required (e.g. 24 hours).
- Also it is impossible to guarantee same order of concurrent transactions.

Recovery via Rollback / Rollforward

- Save results of transactions and when failure occurs to recover

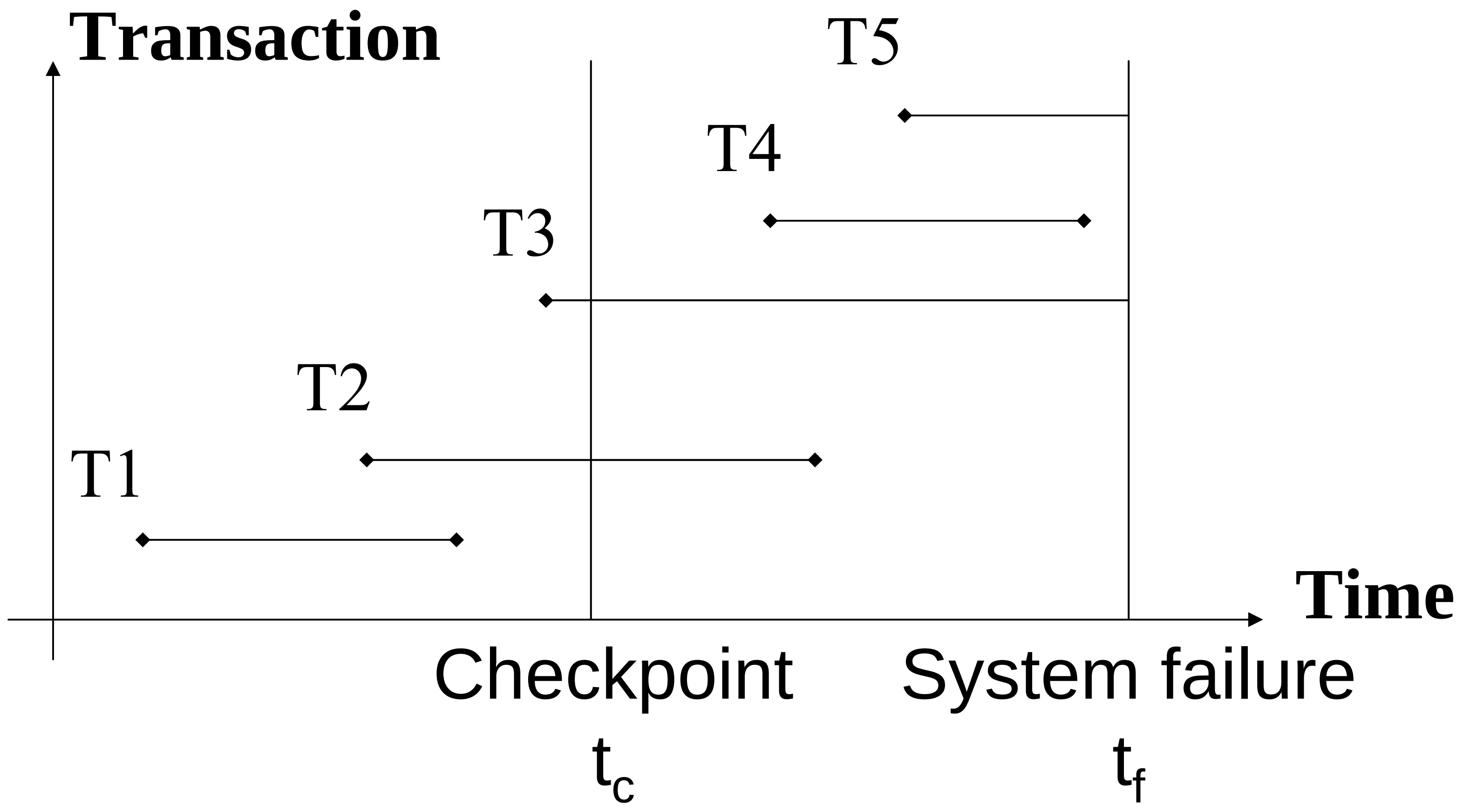
  by removing changes (rollback) then

  reapply the changes (rollforward).
- Here a log is kept. The log contains a record of data changes in chronological order.

Recovery via Rollback / Rollforward

- At certain prescribed intervals. E.g. after specified number of entries have been written to the log the system automatically takes a checkpoint.
  - Physically write the contents of the database buffers out to the physical database.
- Physically write a special checkpoint record out to the physical log. This record gives a list of all transactions that were in progress at that time. i.e. T2-T3

Transactions



Recovery Process

- Recreate (or not destroy) the outputs of all completed transactions.
- Abort all transactions in process at the time of the failure.
- Remove database changes generated by aborted transactions.
- Restart aborted transactions.

When system restarts after a failure

- Using the checkpoint record identify all transactions that were in progress at that time.

  UNDO={T2, T3}. Initial REDO list is empty. REDO={}.
- Search forward through the log starting from the checkpoint record.
- If a “start” log entry is found for transaction T, add T to the UNDO list.

  E.g. T4, T5. UNDO={T2, T3, T4, T5}.

- If a “commit” log entry is found for transaction T **move** T from the UNDO list to the REDO list.

  E.g. T2, T4. **UNDO**={T3, T5}, REDO={T2, T4}.
- When end of the log is reached, the UNDO and REDO lists are identified.
- System now works backwards through the log, undoing the transactions in the UNDO list and then it works forward again redoing the transactions in the REDO list.

  i.e. rollback and rollforward.

Recovery via Rollback / Rollforward

Possible data items of a log record: relative record no, transaction id, reverse pointer, forward pointer, time, type of operation, object, old values, new value.

1	OT1	0	2	11.42	START			
2	OT1	1	4	11.43	MODIFY	CUST143	Old	New
3	OT2	0	8	11.46	START			
4	OT1	2	5	11.47	MODIFY	SPAA	Old	New
5	OT1	4	7	11.47	INSERT	ORDER11		Value
6	CT1	0	9	11.48	START			
7	OT1	5	0	11.49	COMMIT			
8	OT2	3	0	11.50	COMMIT			
9	CT1	6	10	11.51	MODIFY	SPBB	Old	New
10	CT1	9	0	11.51	COMMIT			

Log instances for OT1, OT2, CT1 transactions. Write-ahead log is maintained.

Recovery outline

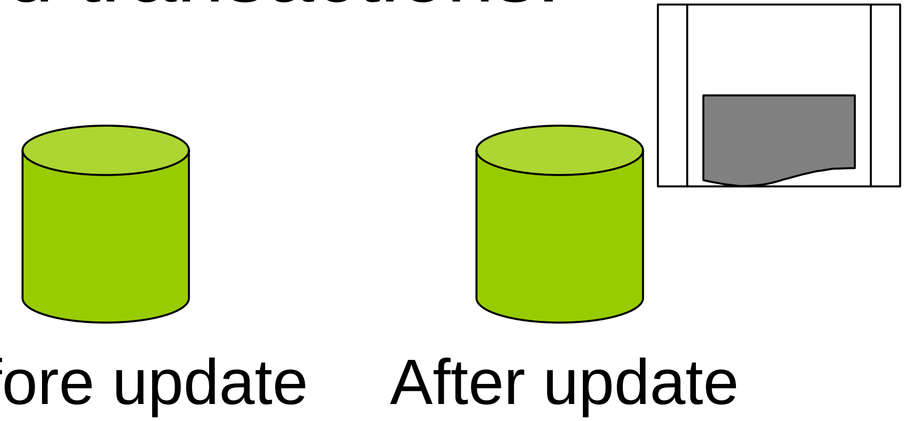
- Recovery from transaction failures usually means that the database is restored to some state from the past so that correct state – close to the time of failure – can be reconstructed from the past state.

The system recovery activity is carried out as part of the system's restart procedure.

Three main techniques for recovery from failures: deferred update, immediate update, shadow paging

Deferred Update

- Do not update the database until after a transaction reaches its commit point.
- Then updates are recorded in the database.
- If transaction fails to reach commit it will not have changed the database in any way - no need to undo the failed transactions.



Deferred Update

	Transactions		Log	Database	
				A=1000; B=2000; C=700	
T1	READ(A)		<T1 start>		
	A = A-50		<T1, A, 950>		
	WRITE(A)	⇨	<T1, B, 2050>		A=1000; B=2000
	READ(B)		<T1 commit>	A=950; B=2050	
	B = B+50		<T2 start>		
	WRITE(B)	⇨	<T2, C, 600>		A=950; B=2050; C=700
T2	READ(C)	⇨	<T2 commit>	C=600	A=950; B=2050; C=600
	C = C-100				
	WRITE(C)				

Update database when <COMMIT>

If fails at ⇨ no REDO/UNDO required

REDO needed as some changes may not have been recorded

From what point to REDO?

Deferred Update with Checkpoint

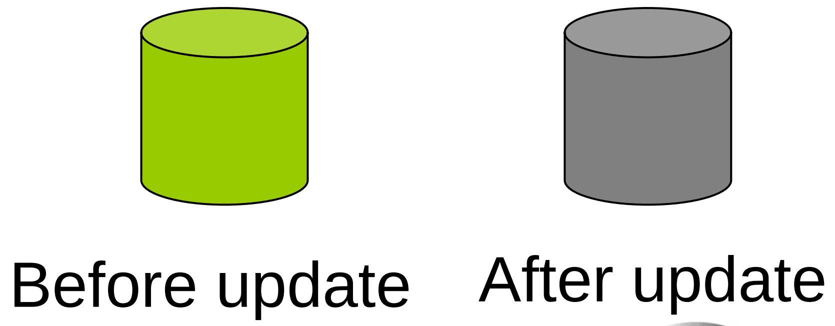
	Log	Database	
	<T0 commit>		
	<T1 start>		
	<checkpoint T1>	A=1000; B=2000; C=700	
	<T1, A, 950>		
⇨	<T1, B, 2050>	A=950; B=2050	A=1000; B=2000
	<T1 commit>		
	<T2 start>		
⇨	<T2, C, 600>	C=600	A=950; B=2050; C=700
⇨	<T2 commit>		A=950; B=2050; C=600
	<T3 start>		

Update database when <CHECKPOINT>

If fails at ⇨ need to REDO/UNDO from CHECKPOINT

Immediate Update

- Database may be updated by some operations of a transaction before the transaction reaches its commit point.
- These operations are typically recorded in the log on disk by force-write before they are applied to the database.
- If a transaction fails the effect of its operations must be undone.



Immediate Update

	Log	**Database**	
		A=1000; B=2000; C=700	
	<T1 start>		
	<T1, A, 1000, 950>	A=950	
⇨	<T1, B, 2000, 2050>	B=2050	A=1000; B=2000
	<T1 commit>		
	<T2 start>		
⇨	<T2, C, 700, 600>	C=600	A=950; B=2050; C=700
⇨	<T2 commit>		A=950; B=2050; C=600

Update database when <WRITE>

If fails at ⇨ need to UNDO, but how far?

Immediate Update with Checkpoint

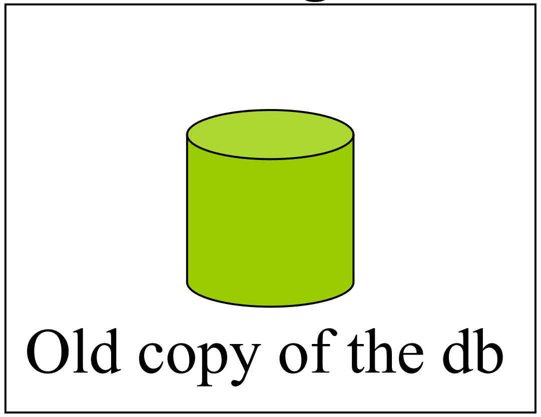
	Log	**Database**	
	<T0 commit>	A=1000; B=2000; C=700	
	<T1 start>		
	<checkpoint T1>		
	<T1, A, 1000, 950>	A=950	
⇨	<T1, B, 2000, 2050>	B=2050	A=1000; B=2000
	<T1 commit>		
	<T2 start>		
⇨	<T2, C, 700, 600>	C=600	A=950; B=2050; C=700
⇨	<T2 commit>		A=950; B=2050; C=600
	<T3 start>		

Also Update database when <CHECKPOINT>

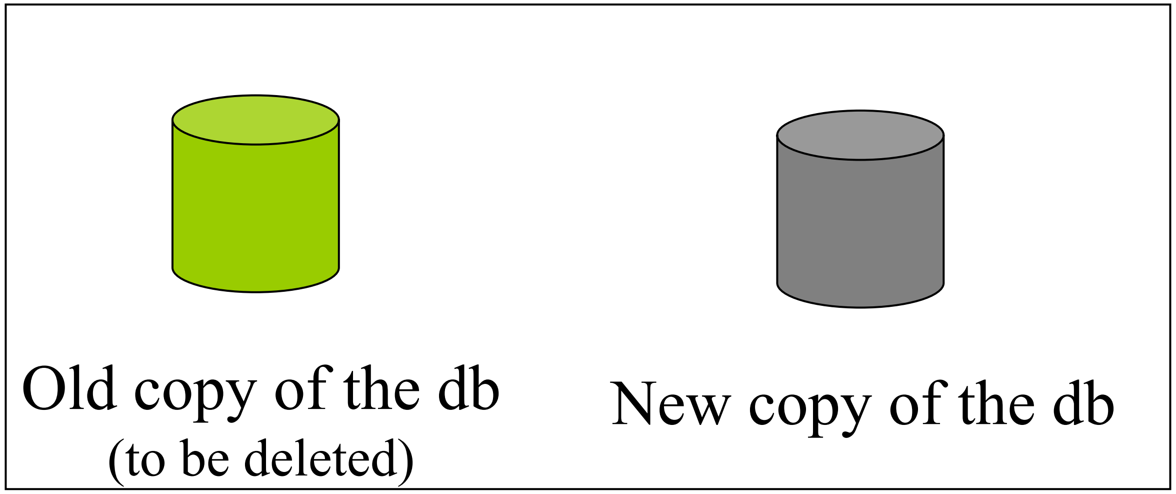
If fails at ⇨ need to REDO/UNDO

Shadow Paging

- The database management system keeps more than one copy of a data item on disk.
- No need to undo a failed transaction, as the original copy of the data is not lost or changed.



Before update



After update

Multi-version

- Reads are never delayed. Reads never delay updates.
  - *if T2 asks for Read(X) when T1 has write(X) then T2 is given access to previously committed version of X;*
  - *if T2 asks for Write(X) when T1 has Read(X) then T2 is given access to X*
- It is never necessary to rollback a read-only transaction
- Deadlock is possible only between update transactions
  - *If T2 asks for Write(X) when T1 has Write(X) then T2 goes to wait state*