CONCURRENCY & RECOVERY

CHAPTER 21-22.1, 23 (6/E)

CHAPTER 17-18.1, 19 (5/E)

LECTURE OUTLINE

- Concurrency
  - Errors in the absence of concurrency control
    - Need to constrain how transactions interleave
  - Goal: Preserve *Isolation* of ACID properties
    - Serializability
  - Two-phase locking
- Reliability & Recovery
  - Errors in the absence of reliability
  - Goal: Preserve *Atomicity* and *Durability* of ACID properties
  - Types of Failures
  - Transaction logs
  - Recovery procedure

LOST UPDATE PROBLEM

- Problematic interleaving of transactions

DB Values	T1		T2	
X = 80				
	read_item(X);	X = 80		
	X := X – 5;	X = 75		
			read_item(X);	X = 80
			X := X + 10;	X = 90
X = 75	write_item(X);			
X = 90			write_item(X);	

- X should be $X_0 - 5 + 10 = 85$
- Occurs when two transactions update the same data item, but both read the same original value before update

$$\ldots\ r_1(X);\ldots;\ r_2(X);\ \ldots;\ w_1(X);\ \ldots;\ w_2(X)$$

$$\ldots\ r_2(X);\ldots;\ r_1(X);\ \ldots;\ w_1(X);\ \ldots;\ w_2(X)$$

DIRTY READ PROBLEM

- *Phantom* update

DB Values	T1		T2	
X = 80				
	read_item(X);	X = 80		
	X := X – 5;	X = 75		
X = 75	write_item(X);			
			read_item(X);	X = 75
			X := X + 10;	X = 85
	X := X / 0;	T1 aborts		
X = 85			write_item(X);	

- X should be as if T_1 didn't execute at all: $X_0 + 10 = 90$
- Occurs when one transaction updates a database item, which is read by another transaction but then the first transaction fails

… $w_1(X)$;…; $r_2(X)$; …; t_1 rolled back

INCONSISTENT READS PROBLEM

- Transactions should read consistent values for isolated state of DB

DB Values	T1		T2	
X = <80, 15, 25>				
			read_item(X_1);	$X_1 = 80$
			SUM := X_1;	SUM = 80
			read_item(X_2);	$X_2 = 15$
			SUM := SUM+X_2;	SUM = 95
	read_item(X_1);	$X_1 = 80$		
	$X_1 := X_1 + 5$;	$X_1 = 85$		
X = <85, 15, 25>	write_item(X_1);			
	read_item(X_3);	$X_3 = 25$		
	$X_3 := X_3 + 5$;	$X_3 = 30$		
X = <85, 15, 30>	write_item(X_3);			
			read_item(X_3);	$X_3= 30$
			SUM := SUM+X_3;	SUM = 125

- SUM should be either 120 (80+15+25, before T_1) or 130 (85+15+30, after T_1)

… $r_2(X)$; …; $w_1(X)$; …; $w_1(Y)$; …; $r_2(Y)$; …

UNREPEATABLE READ PROBLEM

- Even with only one update, might read inconsistent values

DB Values	T1		T2	
X = 80				
			read_item(X);	X = 80
			Y := f(X);	
	read_item(X);	X = 80		
	X := X – 5;	X = 75		
X = 75	write_item(X);			
			read_item(X);	X = 75
			Z := f2(X,Y);	

- Z has a value that depends on two *different* values of X!
- Occurs when one transaction updates a database item, which is read by another transaction both before and after the update

$$\ldots r_2(X);\ \ldots\ w_1(X);\ldots;\ r_2(X);\ \ldots$$

HIGH LEVEL LESSON

- We need to worry about interaction between two applications when
  - one *reads* from the database while the other *writes* to (modifies) the database;
  - both *write* to (modify) the database.
- We do **not** worry about interaction between two applications when both only *read* from the database.

SCHEDULE

- Sequence of interleaved operations from several transactions

	at ATM window #1	at ATM window #2
1	read_item(savings);	
2	savings = savings - $100;	
3		read_item(chequing);
4	write_item(savings);	
5	read_item(chequing);	
6		chequing = chequing - $20;
7		write_item(chequing);
8	chequing = chequing + $100;	
9	write_item(chequing);	
10		dispense $20 to customer;

$$\equiv \quad b_1;\ r_1(s);\ b_2;\ r_2(c);\ w_1(s);\ r_1(c);\ w_2(c);\ w_1(c);\ e_1;\ e_2;$$

SERIAL SCHEDULES

- A schedule S is **serial** if *no interleaving* of operations from several transactions
  - For every transaction T, all the operations of T are executed consecutively
- Assume consistency preservation (ACID property):
  - Each transaction, if executed on its own (from start to finish), will transform a consistent state of the database into another consistent state.
  - Hence, each transaction is correct on its own.
  - Thus, any serial schedule will produce a correct result.
- Serial schedules are not feasible for performance reasons:
  - Long transactions force other transactions to wait
  - When a transaction is waiting for disk I/O or any other event, system cannot switch to other transaction
  - Solution: allow some interleaving

ACCEPTABLE INTERLEAVINGS

- Need to allow interleaving without sacrificing correctness
- Executing some operations in another order causes a different outcome
  - …$r_1(X)$; $w_2(X)$… *vs.* …$w_2(X)$; $r_1(X)$…
    - T1 will read a different value for X
  - …$w_1(Y)$; $w_2(Y)$… *vs.* …$w_2(Y)$; $w_1(Y)$...
    - DB value for Y after both operations will be different
- Two operations **conflict** if:
  1. They access the same data item X
  2. They are from two different transactions
  3. At least one is a write operation
     - Read-Write conflict : … r_1 (X); …; $w_2(X)$; …
     - Write-Write conflict : … $w_1(Y)$; …; $w_2(Y)$; …
- Note that two read operations do *not* conflict.
  - …$r_1(Z)$; $r_2(Z)$… *vs.* …$r_2(Z)$; $r_1(Z)$...
    - both transactions read the same values of Z
- Two schedules are **conflict equivalent** if the relative order of any two *conflicting* operations is the same in both schedules.

SERIALIZABLE SCHEDULES

- Although any serial schedule will produce a correct result, they might not all produce the *same* result.
  - If two people try to reserve the last seat on a plane, only one gets it. The serial order determines which one. The two orderings have different results, but either one is correct.
  - There are *n*! serial schedules for *n* transactions; any of them gives a correct result.
- A schedule S with *n* transactions is **serializable** if it is conflict equivalent to *some* serial schedule of the same *n* transactions.
- Serializable schedule “correct” because equivalent to some serial schedule, and any serial schedule acceptable.
  - It will leave the database in a consistent state.
  - Interleaving such that
    - transactions see data as if they were serially executed
    - transactions leave DB state as if they were serially executed
    - efficiency achievable through concurrent execution

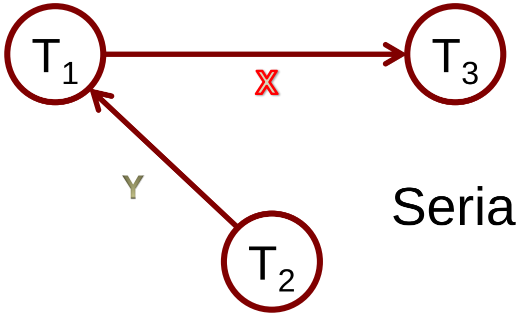
TESTING CONFLICT SERIALIZABILITY

- Consider all read_item and write_item operations in a schedule

1. Construct **serialization** graph
   - Node for each transaction T
   - Directed edge from T_i to T_j if some operation in T_i appears before a conflicting operation in T_j
2. The schedule is serializable if and only if the serialization graph has no cycles.

- Is the following schedule serializable?

b_1; $r_1(X)$; b_2; $r_2(Y)$; $w_1(X)$; b_3; $w_2(Y)$; e_2; $r_1(Y)$; $r_3(X)$; e_3; $w_1(Y)$; e_1;



Serializable; equivalent to: $T_2 \rightarrow T_1 \rightarrow T_3$

b_2; $r_2(Y)$; $w_2(Y)$; e_2; b_1; $r_1(X)$; $w_1(X)$; $r_1(Y)$; $w_1(Y)$; e_1; b_3; $r_3(X)$; e_3;

DATABASE LOCKS

- Use **locks** to ensure that conflicting operations cannot occur
  - **exclusive** lock for writing; **shared** lock for reading
  - cannot read item with first getting shared or exclusive lock on it
  - cannot write item with first getting write (exclusive) lock on it
- Request for lock might cause transaction to **block** (wait)
  - No lock granted on X if some transaction holds write lock on X
    - write lock is exclusive
  - Write lock cannot be granted on X if some transaction holds any lock on X

T1 \ T2	holds read (shared) lock	holds write (exclusive) lock
requests read lock	OK	block T1
requests write lock	block T1	block T1

- Blocked transactions are unblocked and granted the requested lock when conflicting transaction(s) release their lock(s)
  - Like passing a microphone (but two types: one allows sharing)

ENFORCING CONFLICT SERIALIZABILITY

- **Rigorous two-phase locking (2PL)**:
  - Obtain read lock on X if transaction will read X
  - Obtain write lock on X (or promote read lock to write lock) if transaction will write X
  - Release all locks at end of transaction
    - whether commit or abort
  - This is SQL's protocol.
- Rigourous 2PL ensures conflict serializability
- Potential problems:
  - **Deadlock**: T_1 waits for T_2 waits for … waits for T_n waits for T_1
    - Requires assassin
  - **Starvation**: T waits for write lock and other transactions repeatedly grab read locks before all read locks released
    - Requires scheduler

T1	T2
request_read(A);	
read_lock(A);	
read_item(A);	
A := A + 100;	
request_write(A);	
write_lock(A);	
write_item(A);	
	request_read(A);
request_read(B);	
read_lock(B);	
read_item(B);	
B := B -10;	
request_write(B);	
write_lock(B);	
write_item(B);	
commit; /*unlock(A,B)*/	
	read_lock(A);
	read_item(A);
	…

PURPOSE OF DATABASE RECOVERY

- To bring the database into the most recent consistent state that existed prior to a failure
- Goal: preserve ACID properties

  Atomicity, Consistency, Isolation and *Durability*

  - abort (and restart) transactions active at time of failure
  - ensure changes made by committed transactions are not lost
- Complication due to DB execution model:
  - Data items packed into I/O blocks (pages)
  - Updated data first stored in DB cache (at time of write)
  - Actually written to disk (flushed) sometime later

POSSIBLE PROBLEMS

- Consider a transaction that transfer funds from one account (X) to another (Y)

Correct Execution

DB Values	T
X = 80; Y = 100	
	read_item(X);
	X := X – 40;
X = 40; Y = 100	write_item(X);
	read_item(Y);
	Y := Y + 40;
X = 40; Y = 140	write_item(Y);

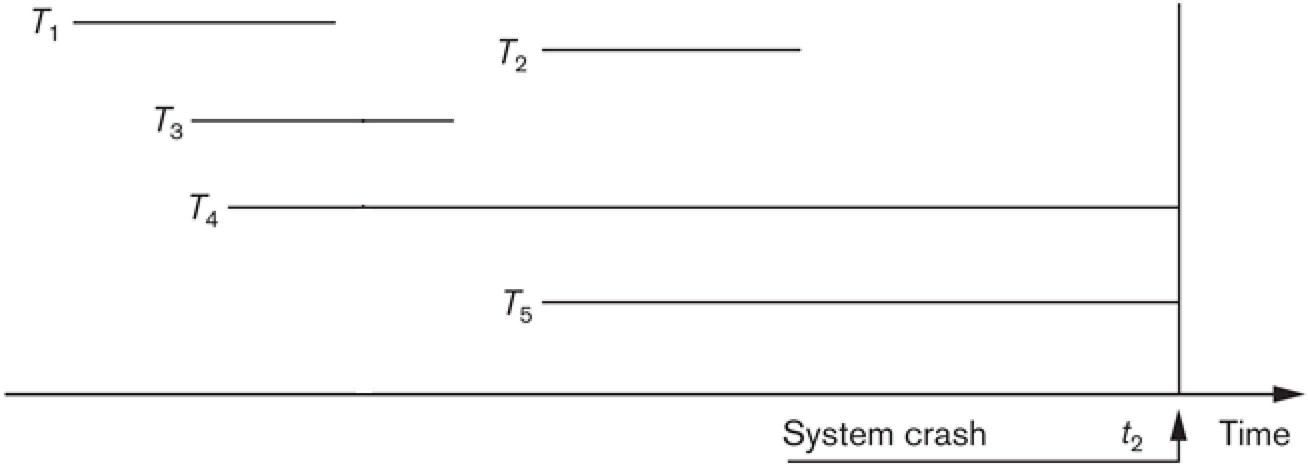
Incorrect Execution

DB Values	T
X = 80; Y = 100	
	read_item(X);
	X := X – 40;
X = 40; Y = 100	write_item(X);
	SYSTEM CRASH!
X = 40; Y = 100	

- High level lesson:
  - We need to worry about partial results of applications on the database when a crash occurs.

PROBLEM SITUATION

- How can we recover from a system crash?
  - DB files preserved but in-memory data lost
    - Contents of data buffers lost
    - Executing programs' states unknown
  - T_1, T_2, T_3 have committed
  - T_4, T_5 still in progress
  - Any of the transactions might have written data
  - Some (unknown) subset of the writes have been flushed to disk



CAUSES OF FAILURE

- Database may become unavailable for use due to
  - **Transaction failure**
    - Incorrect input, deadlock, incorrect synchronization
    - Result: transaction *abort*
  - **System failure**
    - Addressing error, application error, operating system fault, etc.
  - **Media failure**
    - RAM failure, disk head crash, power disruption, etc.
- We wish to recover from system failure.
  - The database server is halted abruptly.
  - Processing of in-progress SQL command(s) is halted abruptly.
  - Connections to application programs (clients) are broken.
  - Contents of memory buffers are lost.
  - Database files are *not* damaged.
    - Recovery from media failure similar, but may need to restore database files from **backup**

KEEP A SYSTEM LOG FILE

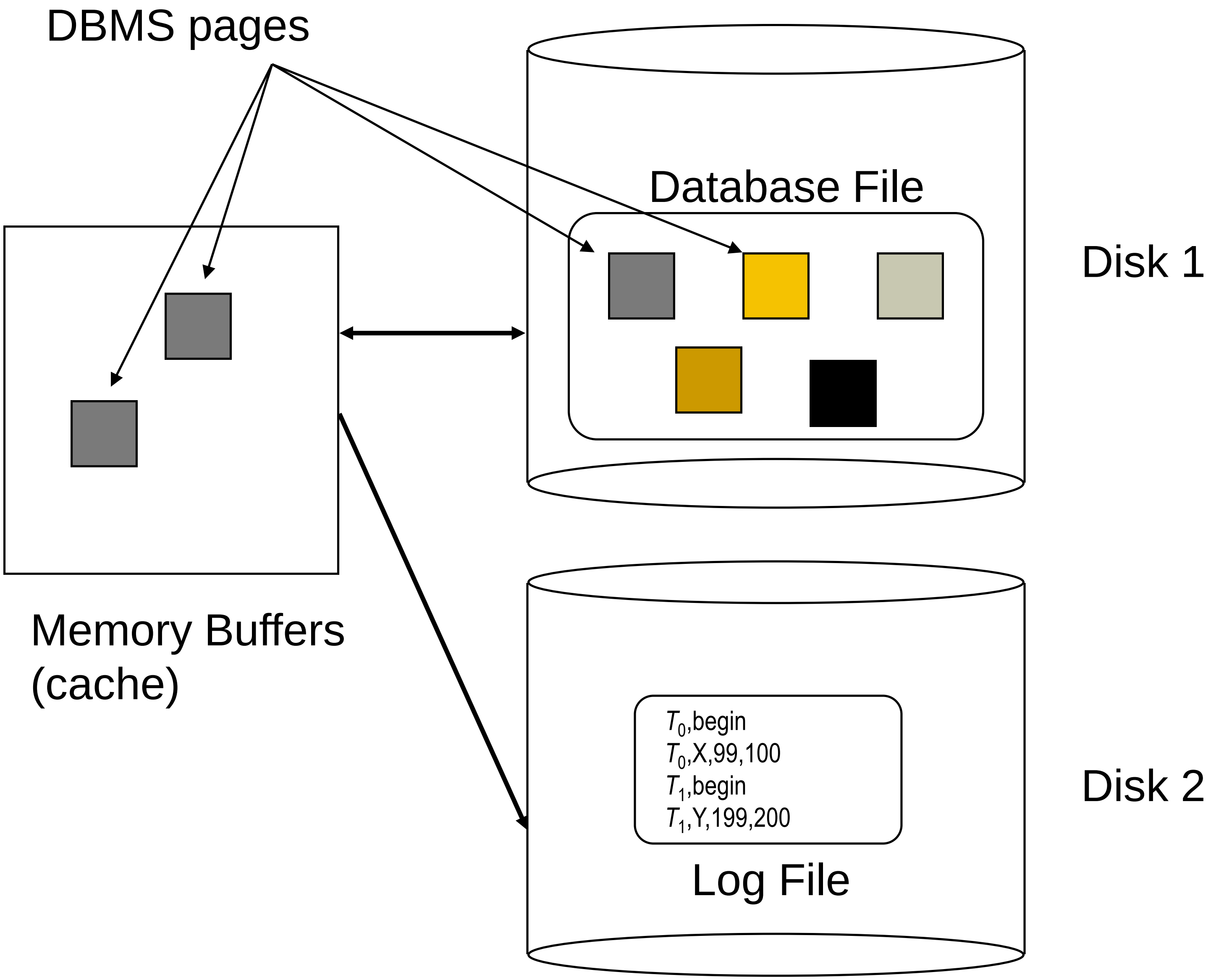
- Append-only file
  - Keep track of all operations of all transactions
  - In the order in which operations occurred
- Stored on disk
  - Persistent except for disk or catastrophic failure
  - Periodically backed up
    - Guard against disk and catastrophic failures
- Main memory buffer
  - Holds records being appended
  - Occasionally whole buffer appended to end of log on disk (flush)

SYSTEM LOG RECORDS

- [**start_transaction**, T]
  - Transaction T has started execution.
- [**write_item**, T, X, old_value, new_value]
  - T has changed the value of item X from old_value to new_value.
  - Before Image (old_value) needed to **undo(X)**
  - After Image (new_value) needed to **redo(X)**
- [**commit**, T]
  - T has completed successfully and committed
  - T's effects (writes) must be durable
- [**abort**, T]
  - T has been aborted
  - T's effects (writes) must be ignored and undone

- *Note*: [**read_item**, T, X] not needed if schedules guaranteed to be *recoverable* (values read must have been committed)

STORAGE STRUCTURE



DBMS pages

Database File

Disk 1

Memory Buffers
(cache)

T_0,begin
T_0,X,99,100
T_1,begin
T_1,Y,199,200

Log File

Disk 2

WRITE-AHEAD LOGGING

- Used to ensure that the log is consistent with the database & to ensure that the log can be used to recover the database to a consistent state
- Two rules:
  1. Log record for a page must be written before corresponding page is flushed to disk, and
  2. All log records must be written before commit.
- A transaction is said to be **committed** when (a) all of its operations are executed, and (b) all its log records are flushed to disk.
- Rule 1 for atomicity
  - so that each operation is known and can be undone if necessary
- Rule 2 for durability
  - so that the effect of a committed transaction is known

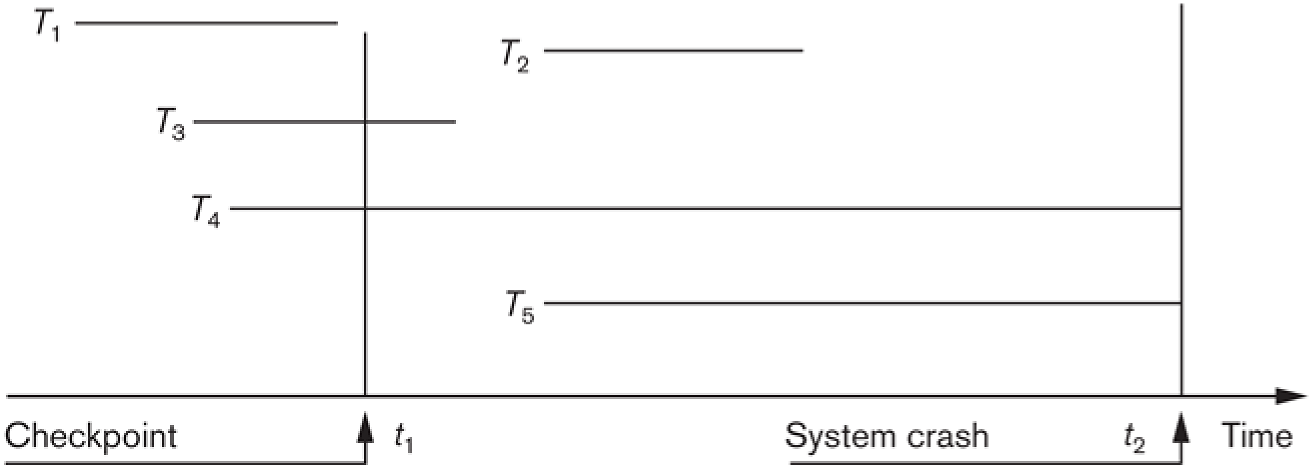
RECOVERY PROCESS

1. Roll-back (undo)
   - Scan log from tail to head (backward in time)
     - create a list of committed transactions
     - create a list of rolled-back transactions
     - undo updates of active transactions
       1. Restore *before image*
       2. Append [undo] record to log (in case of crash *during* recovery)
2. Roll-forward (redo)
   - Scan the log from head to tail (forwards in time)
     - Redo updates of committed transactions
       - Use *after image* for new values
3. Restart executing all in-progress transactions (maybe)

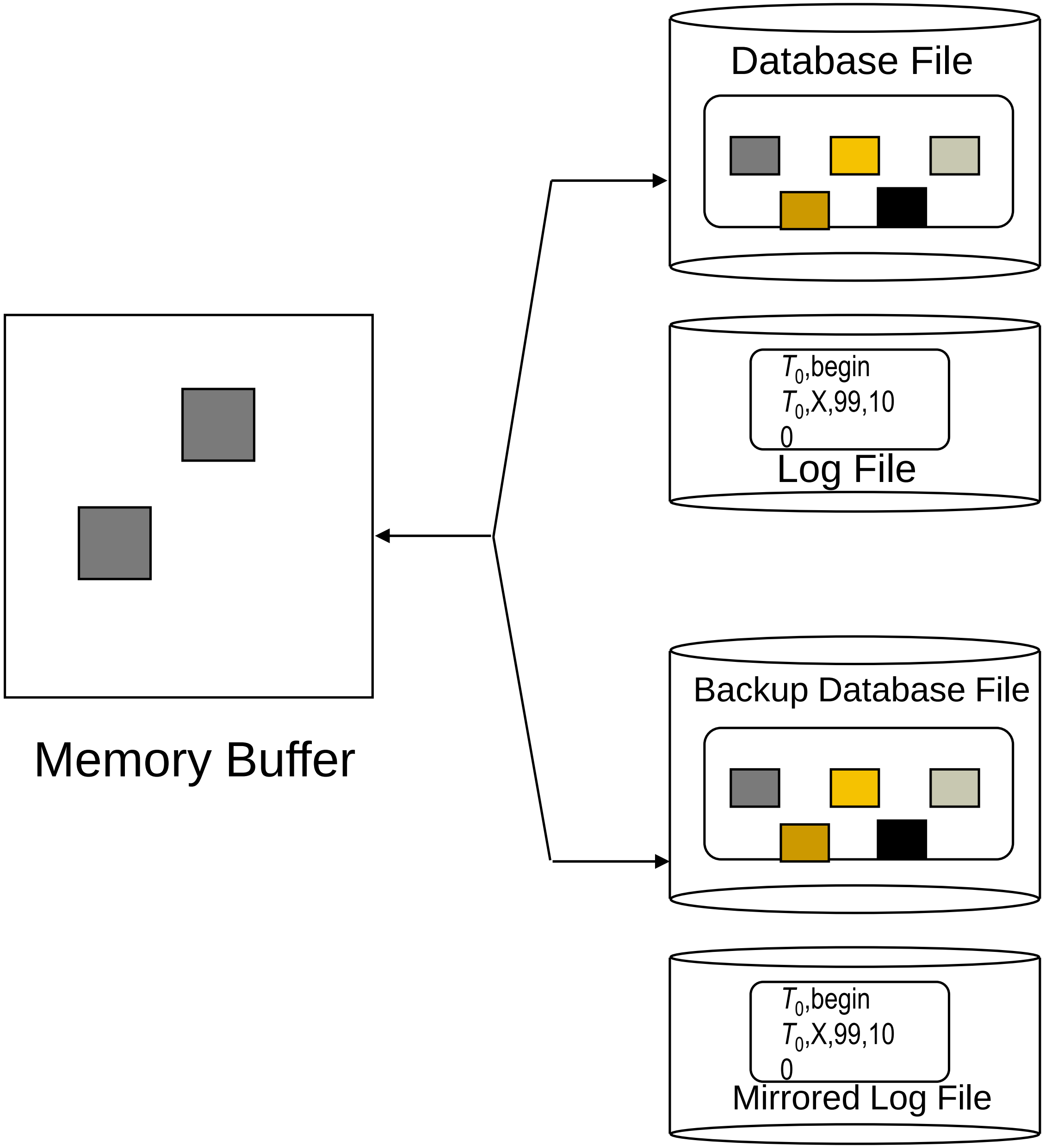
   (those neither committed nor aborted)

CHECKPOINTING

- To save redo effort, use **checkpoints**
  - Occasionally flush data buffers
    1. Suspend execution of transactions temporarily.
    2. Force-write modified (dirty) buffer data to disk.
    3. Append [checkpoint] record to log.
    4. Flush log to disk.
    5. Resume normal transaction execution.
  - During recovery, redo required only for log records appearing after [checkpoint] record



BACKUPS AND MIRRORING



Database File

T_0,begin
T_0,X,99,100
Log File

Memory Buffer

Backup Database File

T_0,begin
T_0,X,99,100
Mirrored Log File

RECOVERY FROM MEDIA FAILURE

1. Restore database from backup
2. Use log to determine which transactions had been committed since the backup
3. Redo committed transaction database updates

LECTURE SUMMARY

- Characterizing schedules based on serializability
  - Serial and non-serial schedules
  - Conflict equivalence of schedules
  - Serialization graph
- Two-phase locking
  - Guarantees conflict serializability
  - Deadlock and starvation
- Databases Recovery
  - Types of Failure
  - Transaction Log
  - Transaction Roll-back (Undo) and Roll-Forward (Redo)
  - Checkpointing