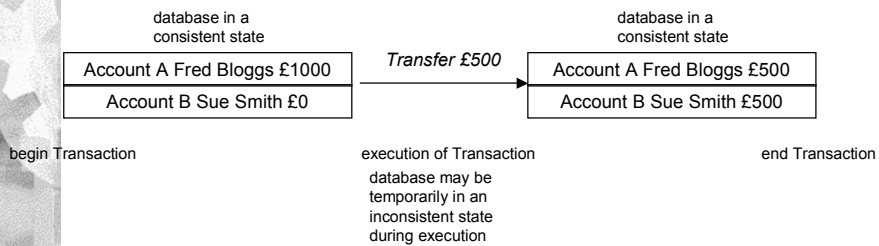


Transaction Processing Recovery & Concurrency Control

What is a transaction

- A transaction is the basic logical unit of execution in an information system. A transaction is a sequence of operations that must be executed as a whole, taking a consistent (& correct) database state into another consistent (& correct) database state;
- A collection of actions that make consistent transformations of system states while preserving system consistency
- An indivisible unit of processing



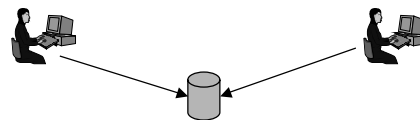
Desirable Properties of ACID Transactions

- A *Atomicity*: a transaction is an atomic unit of processing and it is either performed entirely or not at all
- C *Consistency Preservation*: a transaction's correct execution must take the database from one correct state to another
- I *Isolation/Independence*: the updates of a transaction must not be made visible to other transactions until it is committed (solves the temporary update problem)
- D *Durability (or Permanency)*: if a transaction changes the database and is committed, the changes must never be lost because of subsequent failure
- o *Serialisability*: transactions are considered serialisable if the effect of running them in an interleaved fashion is equivalent to running them serially in some order

Requirements for Database Consistency

✱ Concurrency Control

- Most DBMS are multi-user systems.
- The concurrent execution of many different transactions submitted by various users must be organised such that each transaction does not interfere with another transaction with one another in a way that produces incorrect results.
- The concurrent execution of transactions must be such that each transaction appears to execute in isolation.



✱ Recovery

- System failures, either hardware or software, must not result in an inconsistent database

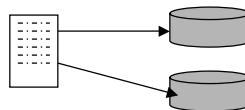
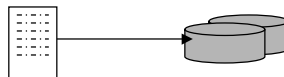
Transaction as a Recovery Unit

- If an error or hardware/software crash occurs between the begin and end, the database will be inconsistent
 - 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
- The database is restored to some state from the past so that a correct state—close to the time of failure—can be reconstructed from the past state.
- A DBMS ensures that if a transaction executes some updates and then a failure occurs before the transaction reaches normal termination, then those updates are undone.
- The statements COMMIT and ROLLBACK (or their equivalent) ensure Transaction Atomicity



Recovery

- **Mirroring**
 - keep two copies of the database and maintain them simultaneously
- **Backup**
 - periodically dump the complete state of the database to some form of tertiary storage
- **System Logging**
 - the log keeps track of all transaction operations affecting the values of database items. The log is kept on disk so that it is not affected by failures except for disk and catastrophic failures.



Recovery from Transaction Failures

Catastrophic failure

- Restore a previous copy of the database from archival backup
- Apply transaction log to copy to reconstruct more current state by redoing committed transaction operations up to failure point
- Incremental dump + log each transaction

Non-catastrophic failure

- Reverse the changes that caused the inconsistency by *undoing* the operations and possibly *redoing* legitimate changes which were lost
- The entries kept in the system log are consulted during recovery.
- No need to use the complete archival copy of the database.

Transaction States

- For recovery purposes the system needs to keep track of when a transaction starts, terminates and commits.
- **Begin_Transaction**: marks the beginning of a transaction execution;
- **End_Transaction**: specifies that the read and write operations have ended and marks the end limit of transaction execution (but may be aborted because of concurrency control);
- **Commit_Transaction**: signals a successful end of the transaction. Any 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. Any changes that the transaction may have applied to the database must be undone;
- **Undo**: similar to **ROLLBACK** but it applies to a single operation rather than to a whole transaction;
- **Redo**: specifies that certain transaction operations must be redone to ensure that all the operations of a committed transaction have been applied successfully to the database;

Entries in the System Log

For every transaction a unique transaction-id is generated by the system.

- **[start_transaction, transaction-id]:** the start of execution of the transaction identified by transaction-id
- **[read_item, transaction-id, X]:** the transaction identified by transaction-id reads the value of database item X. Optional in some protocols.
- **[write_item, transaction-id, X, old_value, new_value]:** the transaction identified by transaction-id changes the value of database item X from old_value to new_value
- **[commit, transaction-id]:** the transaction identified by transaction-id has completed all accesses to the database successfully and its effect can be recorded permanently (committed)
- **[abort, transaction-id]:** the transaction identified by transaction-id has been aborted

```

Credit_labmark (sno
NUMBER, cno CHAR, credit
NUMBER)
old_mark NUMBER;
new_mark NUMBER;

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

new_mark := old_mark +
credit;

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

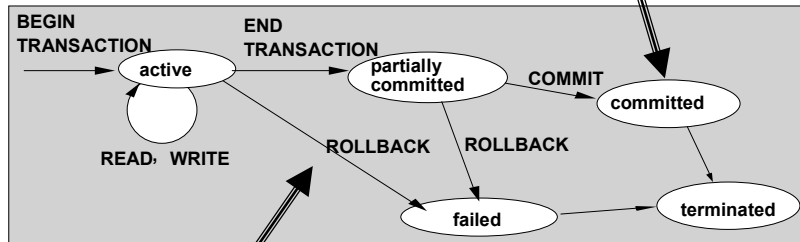
COMMIT;

EXCEPTION
WHEN OTHERS THEN
ROLLBACK;

END credit_labmark;
    
```

Transaction execution

A transaction reaches its *commit point* when all operations accessing the database are completed and the result has been recorded in the log. It then writes a [commit, transaction-id].



If a system failure occurs, searching the log and rollback the transactions that have written into the log a

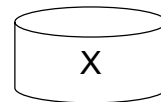
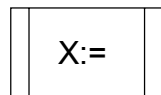
[start_transaction, transaction-id]

[write_item, transaction-id, X, old_value, new_value]

but have not recorded into the log a **[commit, transaction-id]**

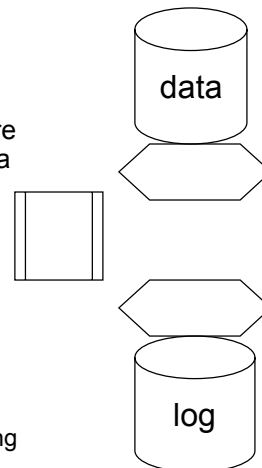
Read and Write Operations of a Transaction

- Specify read or write operations on the database items that are executed as part of a transaction
- **read_item(X):**
 - reads a database item named X into a program variable also named X.
 1. find the address of the disk block that contains item X
 2. copy that disk block into a buffer in the main memory
 3. copy item X from the buffer to the program variable named
- **write_item(X):**
 - writes the value of program variable X into the database item named X.
 1. find the address of the disk block that contains item X
 2. copy that disk block into a buffer in the main memory
 3. copy item X from the program variable named X into its current location in the buffer store the updated block in the buffer back to disk (this step updates the database on disk)



Checkpoints in the System Log

- A [checkpoint] record is written periodically into the log when the system writes out to the database on disk the effect of all WRITE operations of committed transactions.
- All transactions whose [commit, transaction-id] entries can be found in the system log will not require their WRITE operations to be redone in the case of a system crash.
- Before a transaction reaches commit point, force-write or flush the log file to disk before commit transaction.
- **Actions Constituting a Checkpoint**
 - temporary suspension of transaction execution
 - forced writing of all updated database blocks in main memory buffers to disk
 - writing a [checkpoint] record to the log and force writing the log to disk
 - resuming of transaction execution



Write Ahead Logging

“In place” updating protocols: Overwriting data in situ

Deferred Update:

- no actual update of the database until after a transaction reaches its commit point

- Updates recorded in log
- Transaction commit point
- Force log to the disk
- Update the database

FAILURE!
REDO database from log entries
No UNDO necessary because database never altered

Immediate Update:

- the database may be updated by some operations of a transaction before it reaches its commit point.

- Update X recorded in log
- Update X in database
- Update Y recorded in log
- Transaction commit point
- Force log to the disk
- Update Y in database

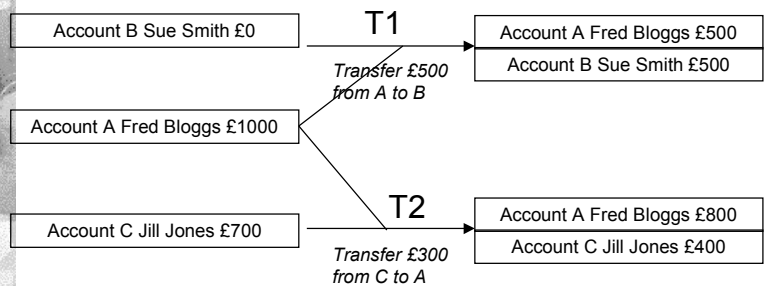
FAILURE!
UNDO X

FAILURE!
REDO Y

- Undo in reverse order in log
- Redo in committed log order
- uses the write_item log entry

Transaction as a Concurrency Unit

- Transactions must be synchronised correctly to guarantee database consistency

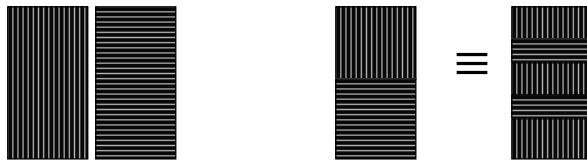


Net result
 Account A 800
 Account B 500
 Account C 400

Transaction scheduling algorithms

- Transaction Serialisability

- The effect on a database of any number of transactions executing in parallel must be the same as if they were executed one after another



- Problems due to the Concurrent Execution of Transactions

- The Lost Update Problem
- The Incorrect Summary or Unrepeatable Read Problem
- The Temporary Update (Dirty Read) Problem

The Lost Update Problem

- Two transactions accessing the same database item have their operations interleaved in a way that makes the database item incorrect

T1: (joe)	T2: (fred)	X	Y
read_item(X);		4	
X := X - N;		2	
	read_item(X);		4
	X := X + M;		7
write_item(X);		2	
read_item(Y);		8	
	write_item(X);		7
Y := Y + N;			10
write_item(Y);			10

X=4
Y=8
N=2
M=3

- item X has incorrect value because its update from T1 is "lost" (overwritten)
- T2 reads the value of X before T1 changes it in the database and hence the updated database value resulting from T1 is lost

The Incorrect Summary or Unrepeatable Read Problem

- 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 others after.

T2 reads X after N is subtracted and reads Y before N is added, so a wrong summary is the result

T1:	T2:	T1	T2	Sum
	sum:= 0;			0
	read_item(A);		4	
	sum:= sum + A;			4
read_item(X);	.	4		
X:= X - N;	.	2		
write_item(X);		2		
	read_item(X);		2	
	sum:= sum + X;			6
	read_item(Y);		8	
	sum:= sum + Y;			14
read_item(Y);		8		
Y:= Y + N;		10		
write_item(Y);		10		

Dirty Read or The Temporary Update Problem

- One transaction updates a database item and then the transaction fails. The updated item is accessed by another transaction before it is changed back to its original value

Joe books seat on flight X

Joe cancels

T1: (joe)	T2: (fred)	Database	Log old	Log new
read_item(X);		4		
X:= X - N;		2		
write_item(X);		2	4	2
	read_item(X);	2	2	
	X:= X- N;	-1		
	write_item(X);	-1	2	-1
failed write (X)		4	rollback T1 log	

Fred books seat on flight X because Joe was on Flight X

- transaction T1 fails and must change the value of X back to its old value
- meanwhile T2 has read the "temporary" incorrect value of X

Schedules of Transactions

- A schedule S of n transactions is a sequential ordering of the operations of the n transactions.
 - *The transactions are interleaved*
- A schedule maintains the order of operations within the individual transaction.
 - For each transaction T if operation a is performed in T before operation b, then operation a will be performed before operation b in S.
 - *The operations are in the same order as they were before the transactions were interleaved*
- Two operations conflict if they belong to different transactions, AND access the same data item AND one of them is a write.

T1

read x
write x

T2

read x
write x

S

read x
read x
write x
write x

Serial and Non-serial Schedules

- A schedule S is *serial* if, for every transaction T participating in the schedule, all of T's operations are executed consecutively in the schedule; otherwise it is called *non-serial*.
- Non-serial schedules mean that transactions are interleaved. There are many possible orders or schedules.
- *Serialisability* theory attempts to determine the 'correctness' of the schedules.
- A schedule S of n transactions is serialisable if it is equivalent to some serial schedule of the same n transactions.

Example of Serial Schedules

• Schedule A

T1: read_item(X); X:= X - N; write_item(X); read_item(Y); Y:=Y + N; write_item(Y);	T2: read_item(X); X:= X + M; write_item(X);
--	--

•Schedule B

T1: read_item(X); X:= X - N; write_item(X); read_item(Y); Y:=Y + N; write_item(Y);	T2: read_item(X); X:= X + M; write_item(X);
--	--

Example of Non-serial Schedules

• Schedule C

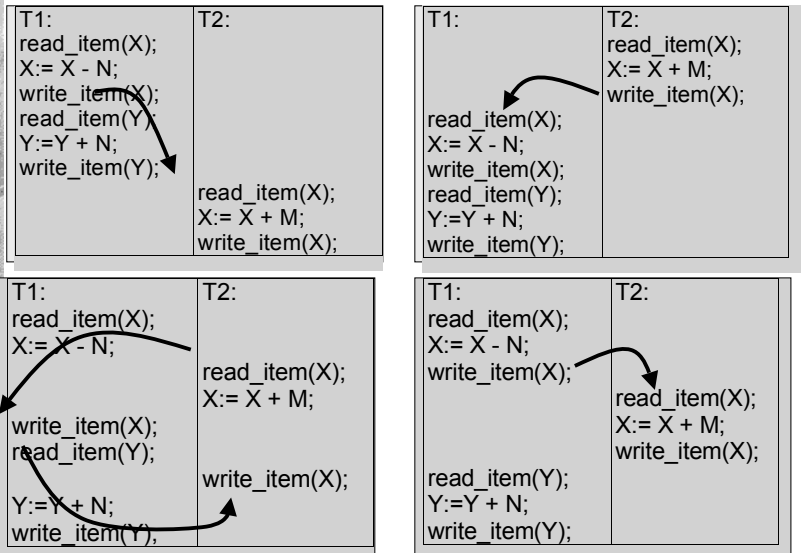
T1: read_item(X); X:= X - N; write_item(X); read_item(Y); Y:=Y + N; write_item(Y);	T2: read_item(X); X:= X + M; write_item(X);
--	--

•Schedule D

T1: read_item(X); X:= X - N; write_item(X); read_item(Y); Y:=Y + N; write_item(Y);	T2: read_item(X); X:= X + M; write_item(X);
--	--

We have to figure out whether a schedule is equivalent to a serial schedule
i.e. the reads and writes are in the right order

Precedence graphs (assuming read X before write X)



View Equivalence and View Serialisability

• View Equivalence:

- As long as each read operation of a transaction reads the result of the same write operation in both schedules, the write operations of each transaction must produce the same results.
- The read operations are said to *see the same view* in both schedules
- The final write operation on each data item is the same in both schedules, so the database state should be the same at the end of both schedules
- A schedule S is view serialisable if it is view equivalent to a serial schedule.
- Testing for view serialisability is NP-complete

Semantic Serialisability

- Some applications can produce schedules that are correct but aren't conflict or view serialisable.
- e.g. Debit/Credit transactions (Addition and subtraction are commutative)

T1	T2
read_item(X);	read_item(Y);
X:=X-10;	Y:=Y-20;
write_item(X);	write_item(Y);
read_item(Y);	read_item(Z);
Y:=Y+10;	Z:=Z+20;
write_item(Y);	write_item(Z);

Schedule

T1	T2
read_item(X);	
X:=X-10;	
write_item(X);	
	read_item(Y);
	Y:=Y-20;
	write_item(Y);
read_item(Y);	
Y:=Y+10;	
write_item(Y);	

Methods for Serialisability

- *Multi-version* Concurrency Control techniques keep the old values of a data item when that item is updated.
- *Timestamps* are unique identifiers for each transaction and are generated by the system. Transactions can then be ordered according to their timestamps to ensure serialisability.
- *Protocols* that, if followed by every transaction, will ensure serialisability of all schedules in which the transactions participate. They may use *locking* techniques of data items to prevent multiple transactions from accessing items concurrently.
- Pessimistic Concurrency Control
 - Check before a database operation is executed by locking data items before they are read and written or checking timestamps

Locking Techniques for Concurrency Control

- The concept of locking data items is one of the main techniques used for controlling the concurrent execution of transactions.
- A lock is a variable associated with a data item in the database. Generally there is a lock for each data item in the database.
- A lock describes the status of the data item with respect to possible operations that can be applied to that item. It is used for synchronising the access by concurrent transactions to the database items.
- A transaction locks an object before using it
- When an object is locked by another transaction, the requesting transaction must wait

Types of Locks

- Binary locks have two possible states:
 1. locked (lock_item(X) operation) and
 2. unlocked (unlock_item(X) operation)
- Multiple-mode locks allow concurrent access to the same item by several transactions. Three possible states:
 1. read locked or shared locked (other transactions are allowed to read the item)
 2. write locked or exclusive locked (a single transaction exclusively holds the lock on the item) and
 3. unlocked.
- Locks are held in a lock table.
 - upgrade lock: read lock to write lock
 - downgrade lock: write lock to read lock

Locks don't guarantee serialisability: Lost Update

T1: (joe)	T2: (fred)	X	Y
write_lock(X)			
read_item(X);		4	
X := X - N;		2	
unlock(X)			
	write_lock(X)		
	read_item(X);	4	
	X := X + M;	7	
	unlock(X)		
write_lock(X)			
write_item(X);		2	
unlock(X)			
write_lock(Y)			
read_item(Y);			8
	write_lock(X)		
Y := Y + N;	write_item(X);	7	
write_item(Y);	unlock(X)		
unlock(Y)			10
			10

Locks don't guarantee serialisability

X=20, Y=30

T1
 read_lock(Y);
 read_item(Y);
 unlock(Y);
 write_lock(X);
 read_item(X);
 X := X + Y;
 write_item(X);
 unlock(X);

T2
 read_lock(X);
 read_item(X);
 unlock(X);
 write_lock(Y);
 read_item(Y);
 Y := X + Y;
 write_item(Y);
 unlock(Y);

Y is unlocked too early

X is unlocked too early

- Schedule 1: T1 followed by T2 \Rightarrow X=50, Y=80
- Schedule 2: T2 followed by T1 \Rightarrow X=70, Y=50

Non-serialisable schedule S that uses locks

X=20
Y=30

T1	T2
read_lock(Y); read_item(Y); unlock(Y);	
	read_lock(X); read_item(X); unlock(X); write_lock(Y); read_item(Y); Y:=X+Y; write_item(Y); unlock(Y);
write_lock(X); read_item(X); X:=X+Y; write_item(X); unlock(X);	

result of S \Rightarrow X=50, Y=50

Ensuring Serialisability: Two-Phase Locking

- All locking operations (read_lock, write_lock) precede the first unlock operation in the transactions.
- Two phases:
 - *expanding phase*: new locks on items can be acquired but none can be released
 - *shrinking phase*: existing locks can be released but no new ones can be acquired

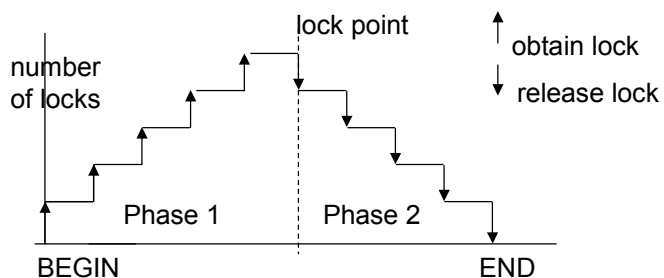
X=20, Y=30

T1	T2
read_lock(Y); read_item(Y); write_lock(X); unlock(Y); read_item(X); X:=X+Y; write_item(X); unlock(X);	read_lock(X); read_item(X); write_lock(Y); unlock(X); read_item(Y); Y:=X+Y; write_item(Y); unlock(Y);

Two-Phasing Locking

- **Basic 2PL**

- When a transaction releases a lock, it may not request another lock

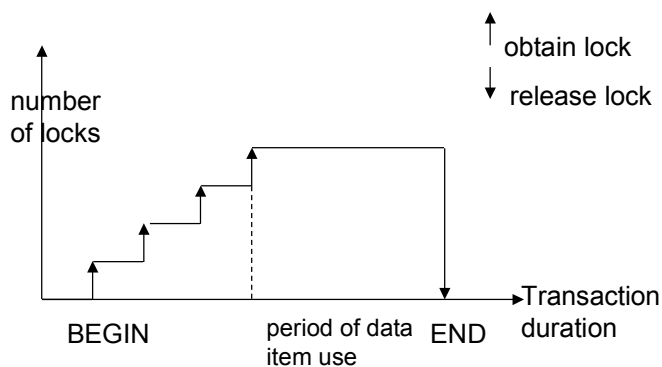


- **Conservative 2PL or static 2PL**

- a transaction locks all the items it accesses before the transaction begins execution
- pre-declaring read and write sets

Two-Phasing Locking

- **Strict 2PL** a transaction does not release any of its locks until after it commits or aborts
- leads to a strict schedule for recovery



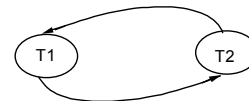
Locking Problems: Deadlock

- Each of two or more transactions is waiting for the other to release an item. Also called a deadly embrace

T1	T2
read_lock(Y); read_item(Y);	read_lock(X); read_item(X);
write_lock(X);	write_lock(Y);

Deadlocks and Livelocks

- Deadlock prevention protocol:
 - conservative 2PL
 - transaction stamping (younger transactions aborted)
 - no waiting
 - cautious waiting
 - time outs
- Deadlock detection (if the transaction load is light or transactions are short and lock only a few items)
 - wait-for graph for deadlock detection
 - victim selection
 - cyclic restarts
- Livelock: a transaction cannot proceed for an indefinite period of time while other transactions in the system continue normally.
 - fair waiting schemes (i.e. first-come-first-served)



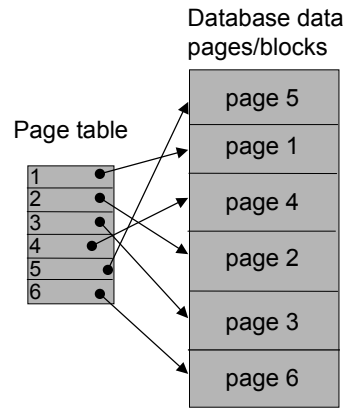
Locking Granularity

- A database item could be
 - a database record
 - a field value of a database record
 - a disk block
 - the whole database
- Trade-offs
 - coarse granularity
 - the larger the data item size, the lower the degree of concurrency
 - fine granularity
 - the smaller the data item size, the more locks to be managed and stored, and the more lock/unlock operations needed.

Other Recovery and Concurrency Strategies

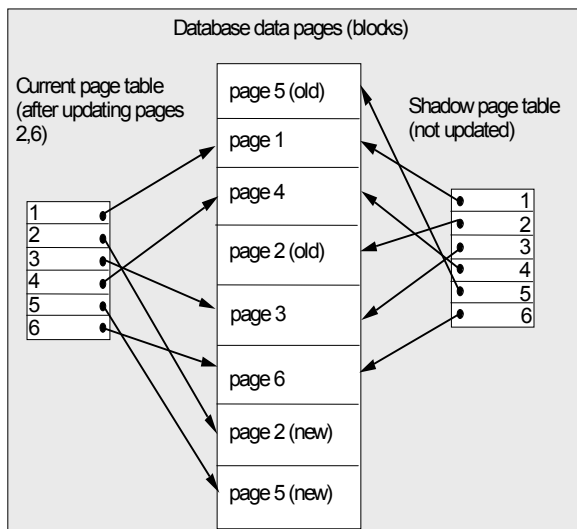
Recovery: Shadow Paging Technique

- Data isn't updated 'in place'
- The database is considered to be made up of a number of n fixed-size disk blocks or pages, for recovery purposes.
- A page table with n entries is constructed where the i^{th} page table entry points to the j^{th} database page on disk.
- Current page table points to most recent current database pages on disk



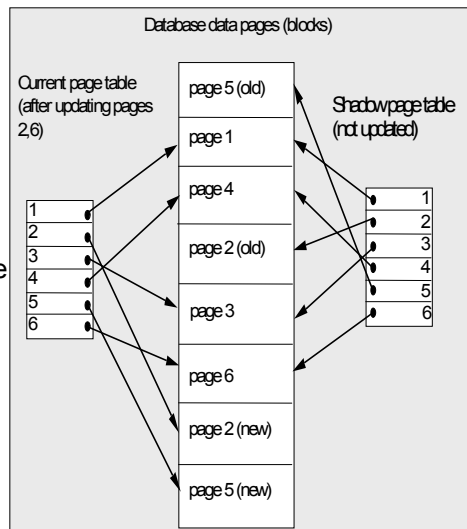
Shadow Paging Technique

- When a transaction begins executing
 - the current page table is copied into a shadow page table
 - shadow page table is then saved
 - shadow page table is never modified during transaction execution
 - writes operations—new copy of database page is created and current page table entry modified to point to new disk page/block



Shadow Paging Technique

- ✱ To recover from a failure
 - ✱ the state of the database before transaction execution is available through the shadow page table
 - ✱ free modified pages
 - ✱ discard current page table
 - ✱ that state is recovered by reinstating the shadow page table to become the current page table once more
- ✱ Committing a transaction
 - ✱ discard previous shadow page
 - ✱ free old page tables that it references
- ✱ Garbage collection



Optimistic Concurrency Control



- ✱ No checking while the transaction is executing.
 - ✱ Check for conflicts after the transaction.
 - ✱ Checks are all made at once, so low transaction execution overhead
 - ✱ Relies on little interference between transactions
 - ✱ Updates are not applied until `end_transaction`
 - ✱ Updates are applied to *local copies* in a transaction space.
1. *read phase*: read from the database, but updates are applied only to local copies
 2. *validation phase*: check to ensure serialisability will not be validated if the transaction updates are actually applied to the database
 3. *write phase*: if validation is successful, transaction updates applied to database; otherwise updates are discarded and transaction is aborted and restarted.

Validation Phase

- Use transaction timestamps
- write_sets and read_sets maintained
- Transaction B is committed or in its validation phase
- Validation Phase for Transaction A
- To check that TransA does not interfere with TransB the following must hold:
 - TransB completes its write phase before TransA starts its reads phase
 - TransA starts its write phase after TransB completes its write phase, and the read set of TransA has no items in common with the write set of TransB
 - Both the read set and the write set of TransA have no items in common with the write set of TransB, and TransB completes its read phase before TransA completes its read phase.

Conclusions

- Transaction management deals with two key requirements of any database system:
- Resilience
 - in the ability of data surviving hardware crashes and software errors without sustaining loss or becoming inconsistent
- Access Control
 - in the ability to permit simultaneous access of data multiple users in a consistent manner and assuring only authorised access