Chapter 6: Transactions

Transaction Concept

1. A ***transaction*** is a *unit* of program execution that accesses and possibly updates various data items.
2. A transaction must see a consistent database.
3. During transaction execution the database may be inconsistent.
4. When the transaction is committed, the database must be consistent.
5. Two main issues to deal with:
   - ★ Failures of various kinds, such as hardware failures and system crashes
   - ★ Concurrent execution of multiple transactions

ACID Properties

To preserve integrity of data, the database system must ensure:

1. **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
2. **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
3. **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
   a. That is, for every pair of transactions *Ti* and *Tj,* it appears to *Ti* that either *Tj,* finished execution before *Ti* started, or *Tj* started execution after *Ti* finished.
4. **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

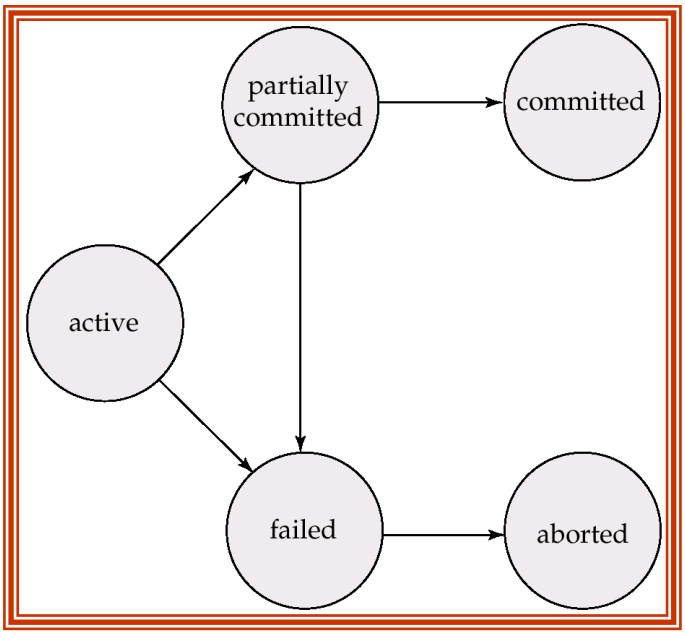
Example of Fund Transfer

1. Transaction to transfer \$50 from account *A* to account *B*:
   read(*A*)
   A := *A* – 50
   write(*A*)
   read(*B*)
   B := *B* + 50
   write(*B)*
2. Consistency requirement – the sum of *A* and *B* is unchanged by the execution of the transaction.
3. Atomicity requirement — if the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, else an inconsistency will result.
4. Durability requirement — once the user has been notified that the transaction has completed (i.e., the transfer of the \$50 has taken place), the updates to the database by the transaction must persist despite failures.
5. Isolation requirement — if between steps 3 and 6, another transaction is allowed to access the partially updated database, it will see an inconsistent database (the sum *A* + *B* will be less than it should be).

Can be ensured trivially by running transactions ***serially***, that is one after the other. However, executing multiple transactions concurrently has significant benefits, as we will see.

Transaction State

1. **Active,** the initial state; the transaction stays in this state while it is executing
2. **Partially committed,** after the final statement has been executed.
3. **Failed,** after the discovery that normal execution can no longer proceed.
4. **Aborted,** after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
   a. restart the transaction – only if no internal logical error
   b. kill the transaction
5. **Committed,** after *successful completion*.



Concurrent Executions

1. Multiple transactions are allowed to run concurrently in the system. Advantages are:
   a. **increased processor and disk utilization**, leading to better transaction *throughput:* one transaction can be using the CPU while another is reading from or writing to the disk
   b. **reduced average response time** for transactions: short transactions need not wait behind long ones.
2. *Concurrency control schemes* – mechanisms to achieve isolation, i.e., to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

Schedules

1. *Schedules* – sequences that indicate the chronological order in which instructions of concurrent transactions are executed
   - ★ a schedule for a set of transactions must consist of all instructions of those transactions
   - ★ must preserve the order in which the instructions appear in each individual transaction.

Example Schedules

1. Let $T1$ transfer \$50 from A to B, and $T2$ transfer 10% of the balance from A to B. The following is a serial schedule (Schedule 1 in the text), in which $T1$ is followed by $T2$.

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

2. Let $T1$ and $T2$ be the transactions defined previously. The following schedule (Schedule 3 in the text) is not a serial schedule, but it is *equivalent* to Schedule 1.

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

In both Schedule 1 and 3, the sum A + B is preserved.

3. The following concurrent schedule (Schedule 4 in the text) does not preserve the value of the the sum $A + B$.

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

Serializability

1. Basic Assumption – Each transaction preserves database consistency.
2. Thus serial execution of a set of transactions preserves database consistency.
3. A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:
   a) conflict serializability

Concurrency Control

Lock-Based Protocols

1. A lock is a mechanism to control concurrent access to a data item
2. Data items can be locked in two modes :
   a. *exclusive (X) mode*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
   b. *shared (S) mode*. Data item can only be read. S-lock is requested using **lock-S** instruction.
3. Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.
4. Lock-compatibility matrix

	S	X
S	true	false
X	false	false

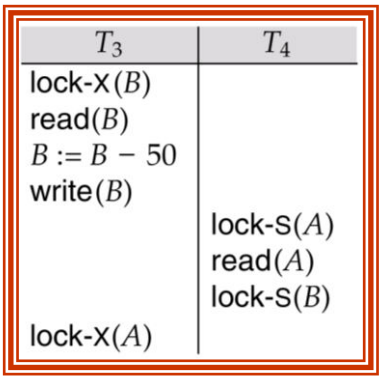
5. A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions

6. Any number of transactions can hold shared locks on an item, but if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.
7. If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.
8. Example of a transaction performing locking:

   T2: **lock-S***(A)*;
   read *(A)*;
   unlock*(A)*;
   lock-S*(B)*;
   read *(B)*;
   unlock*(B)*;
   display*(A+B)*
9. A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

Pitfalls of Lock-Based Protocols

1. Consider the partial schedule



T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

2. Neither *T3* nor *T4* can make progress — executing **lock-S***(B)* causes *T4* to wait for *T3* to release its lock on *B*, while executing **lock-X***(A)* causes *T3* to wait for *T4* to release its lock on *A*.
3. Such a situation is called a **deadlock**.
   a. To handle a deadlock one of *T3* or *T4* must be rolled back and its locks released.
4. The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.
5. **Starvation** is also possible if concurrency control manager is badly designed. For example:
   a. A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
   b. The same transaction is repeatedly rolled back due to deadlocks.
6. Concurrency control manager can be designed to prevent starvation.

The Two-Phase Locking Protocol

1. This is a protocol which ensures conflict-serializable schedules.
2. Phase 1: Growing Phase
    a. transaction may obtain locks
    b. transaction may not release locks
3. Phase 2: Shrinking Phase
    a. transaction may release locks
    b. transaction may not obtain locks
4. The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e. the point where a transaction acquired its final lock
5. Two-phase locking *does not* ensure freedom from deadlocks

Implementation of Locking

1. A **Lock manager** can be implemented as a separate process to which transactions send lock and unlock requests
2. The lock manager replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock)
3. The requesting transaction waits until its request is answered
4. The lock manager maintains a data structure called a **lock table** to record granted locks and pending requests
5. The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked

Timestamp-Based Protocols

1. Each transaction is issued a timestamp when it enters the system. If an old transaction *Ti* has time-stamp TS(*Ti*), a new transaction *Tj* is assigned time-stamp TS(*Tj*) such that TS(*Ti*) <TS(*Tj*).
2. The protocol manages concurrent execution such that the time-stamps determine the serializability order.
3. In order to assure such behavior, the protocol maintains for each data *Q* two timestamp values:
    a. **W-timestamp**(*Q*) is the largest time-stamp of any transaction that executed **write**(*Q*) successfully.
    b. **R-timestamp**(*Q*) is the largest time-stamp of any transaction that executed **read**(*Q*) successfully.
4. The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.
5. Suppose a transaction Ti issues a **read**(*Q*)
    a. If TS(*Ti*) < **W**-timestamp(*Q*), then *Ti* needs to read a value of *Q* that was already overwritten. Hence, the **read** operation is rejected, and *Ti* is rolled back.
    b. If TS(*Ti*)≥ **W**-timestamp(*Q*), then the **read** operation is executed, and R-timestamp(*Q*) is set to the maximum of R-timestamp(*Q*) and TS(*Ti*).
6. Suppose that transaction *Ti* issues **write**(*Q*).

a. If TS(Ti) < R-timestamp(Q), then the value of Q that Ti is producing was needed previously, and the system assumed that that value would never be produced. Hence, the **write** operation is rejected, and Ti is rolled back.
b. If TS(Ti) < W-timestamp(Q), then Ti is attempting to write an obsolete value of Q. Hence, this **write** operation is rejected, and Ti is rolled back.
c. Otherwise, the **write** operation is executed, and W-timestamp(Q) is set to TS(Ti).

Deadlock Handling

1. Consider the following two transactions:
   $T1$: write (X) $T2$: write(Y)
   write(Y) write(X)
2. Schedule with deadlock

T1	**T2**
lock-X on X	
write (X)	
	lock-X on Y
	write (X)
	wait for **lock-X** on X
wait for **lock-X** on Y	

3. System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.
4. ***Deadlock prevention*** protocols ensure that the system will *never* enter into a deadlock state. Some prevention strategies :
   ★ Require that each transaction locks all its data items before it begins execution (predeclaration).
   ★ Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol).

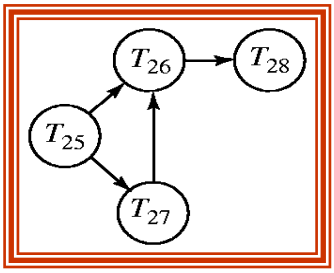
More Deadlock Prevention Strategies

1. Following schemes use transaction timestamps for the sake of deadlock prevention alone.
   ★ **wait-die** scheme — non-preemptive
   older transaction may wait for younger one to release data item. Younger transactions never wait for older ones; they are rolled back instead. A transaction may die several times before acquiring needed data item
   ★ **wound-wait** scheme — preemptive
   older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones. May be fewer rollbacks than *wait-die* scheme.
2. Both in *wait-die* and in *wound-wait* schemes, a rolled back transactions is restarted with its original timestamp. Older transactions thus have precedence over newer ones, and starvation is hence avoided.
3. Timeout-Based Schemes :

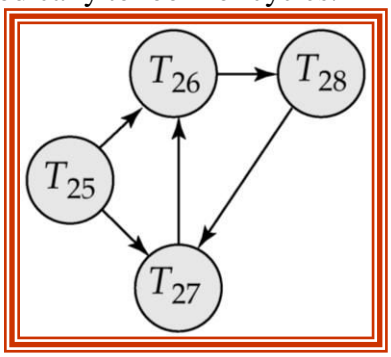
- ★ a transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back.
- ★ thus deadlocks are not possible
- ★ simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval.

Deadlock Detection

1. Deadlocks can be described as a *wait-for graph*, which consists of a pair $G = (V,E)$,
   a. V is a set of vertices (all the transactions in the system)
   b. E is a set of edges; each element is an ordered pair $Ti \rightarrow Tj$.
2. If $Ti \rightarrow Tj$ is in E, then there is a directed edge from Ti to Tj, implying that Ti is waiting for Tj to release a data item.
3. When Ti requests a data item currently being held by Tj, then the edge Ti Tj is inserted in the wait-for graph. This edge is removed only when Tj is no longer holding a data item needed by Ti.
4. The system is in a deadlock state if and only if the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm periodically to look for cycles.



Wait-for graph without a cycle



Wait-for graph with a cycle

Deadlock Recovery

1. When deadlock is detected :
   - ★ Some transaction will have to rolled back (made a victim) to break deadlock. Select that transaction as victim that will incur minimum cost.
   - ★ Rollback -- determine how far to roll back transaction
     - ➢ Total rollback: Abort the transaction and then restart it.
     - ➢ More effective to roll back transaction only as far as necessary to break deadlock.
   - ★ Starvation happens if same transaction is always chosen as victim. Include the number of rollbacks in the cost factor to avoid starvation

Recovery System
Failure Classification

1. **Transaction failure** :
   - ★ **Logical errors**: transaction cannot complete due to some internal error condition

- ★ **System errors**: the database system must terminate an active transaction due to an error condition (e.g., deadlock)

2. **System crash**: a power failure or other hardware or software failure causes the system to crash.
   - ★ **Fail-stop assumption**: non-volatile storage contents are assumed to not be corrupted by system crash
     - ➢ Database systems have numerous integrity checks to prevent corruption of disk data
3. **Disk failure**: a head crash or similar disk failure destroys all or part of disk storage
   - ★ Destruction is assumed to be detectable: disk drives use checksums to detect failures

Recovery and Atomicity

1. Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state.
2. Consider transaction *Ti* that transfers $50 from account *A* to account *B*; goal is either to perform all database modifications made by *Ti* or none at all.
3. Several output operations may be required for *Ti* (to output *A* and *B*). A failure may occur after one of these modifications have been made but before all of them are made.
4. To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.
5. Two approaches are:
   - ★ **log-based recovery**, and
   - ★ **shadow-paging**

Log-Based Recovery

1. A **log** is kept on stable storage.
   a. The log is a sequence of **log records**, and maintains a record of update activities on the database.
2. When transaction *Ti* starts, it registers itself by writing a
   <*Ti* **start**>log record
3. *Before Ti* executes **write**(*X*), a log record <*Ti, X, V1, V2*> is written, where *V1* is the value of *X* before the write, and *V2* is the value to be written to *X*.
   a. Log record notes that *Ti* has performed a write on data item *Xj* *Xj* had value *V1* before the write, and will have value *V2* after the write.
4. When *Ti* finishes it last statement, the log record <*Ti* **commi**t> is written.
5. We assume for now that log records are written directly to stable storage (that is, they are not buffered)
6. Two approaches using logs
   a. Deferred database modification
   b. Immediate database modification

Deferred Database Modification

1. The **deferred database modification** scheme records all modifications to the log, but defers all the **write**s to after partial commit.
2. Assume that transactions execute serially
3. Transaction starts by writing <*Ti* ***start***> record to log.
4. A **write**(*X*) operation results in a log record <*Ti, X, V*> being written, where *V* is the new value for *X*
    a. Note: old value is not needed for this scheme
5. The write is not performed on *X* at this time, but is deferred.
6. When *Ti* partially commits, <*Ti* **commit**> is written to the log
7. Finally, the log records are read and used to actually execute the previously deferred writes.
8. During recovery after a crash, a transaction needs to be redone if and only if both <*Ti* **start**> and<*Ti* **commit**> are there in the log.
9. Redoing a transaction *Ti* (**redo***Ti*) sets the value of all data items updated by the transaction to the new values.
10. Crashes can occur while
    a. the transaction is executing the original updates, or
    b. while recovery action is being taken
11. Example transactions *T0* and *T1* (*T0* executes before *T1*):
    [A=1000,B=2000,C=500]

T0: **read** (*A*)	*T1* : **read** (*C*)
A: - A - 50	*C:- C- 100*
Write (*A*)	**write** (*C*)
read (*B*)	
B:- B + 50	
write (*B*)	

12. Below show the log as it appears at three instances of time.

<T_0 start>	<T_0 start>	<T_0 start>
<T_0, *A*, 950>	<T_0, *A*, 950>	<T_0, *A*, 950>
<T_0, *B*, 2050>	<T_0, *B*, 2050>	<T_0, *B*, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, *C*, 600>	<T_1, *C*, 600>
		<T_1 commit>
(a)	(b)	(c)

13. If log on stable storage at time of crash is as in case:
    (a) No redo actions need to be taken
    (b) redo(*T*0) must be performed since <*T*0 **commi**t> is present
    (c) **redo**(*T*0) must be performed followed by redo(*T*1) since
        <*T*0 **commit**> and <*T1* **commit**> are present

Immediate Database Modification

1. The **immediate database modification** scheme allows database updates of an uncommitted transaction to be made as the writes are issued

a. since undoing may be needed, update logs must have both old value and new value

2. Update log record must be written *before* database item is written
   a. We assume that the log record is output directly to stable storage
   b. Can be extended to postpone log record output, so long as prior to execution of an **output**(*B*) operation for a data block B, all log records corresponding to items *B* must be flushed to stable storage
3. Output of updated blocks can take place at any time before or after transaction commit
4. Order in which blocks are output can be different from the order in which they are written.

Example

Log	**Write**
<*T*0 **start**>	
<*T0,* A, 1000, 950>	
<*T0,* B, 2000, 2050>	
	A = 950
	B = 2050
<*T*0 **commit**>	
<*T*1 **start**>	
<*T*1, C, 700, 600>	
	C = 600
<*T*1 **commit**>	

5. Recovery procedure has two operations instead of one:
   a. **undo**(*T*i) restores the value of all data items updated by *Ti* to their old values, going backwards from the last log record for *Ti*
   b. **redo**(*T*i) sets the value of all data items updated by *Ti* to the new values, going forward from the first log record for *Ti*
6. Both operations must be **idempotent**
   a. That is, even if the operation is executed multiple times the effect is the same as if it is executed once
      i. Needed since operations may get re-executed during recovery
7. When recovering after failure:
   a. Transaction *Ti* needs to be undone if the log contains the record <*Ti* **start**>, but does not contain the record <*Ti* **commit**>.
   b. Transaction *Ti* needs to be redone if the log contains both the record <*Ti* **start**> and the record <*Ti* **commit**>.
8. Undo operations are performed first, then redo operations.
9. Below we show the log as it appears at three instances of time

(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>

Recovery actions in each case above are:

(a) undo (*T*0): B is restored to 2000 and A to 1000.

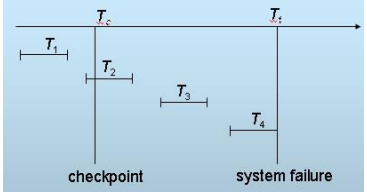
(b) undo (*T*1) and redo (*T*0): C is restored to 700, and then *A* and *B* are set to 950 and 2050 respectively.

(c) redo (*T*0) and redo (*T*1): A and B are set to 950 and 2050 respectively. Then *C* is set to 600

Checkpoints

1. Problems in recovery procedure as discussed earlier :
   a. searching the entire log is time-consuming
   b. we might unnecessarily redo transactions which have already output their updates to the database.
2. Streamline recovery procedure by periodically performing **checkpointing**
   a. Output all log records currently residing in main memory onto stable storage.
   b. Output all modified buffer blocks to the disk.
   c. Write a log record < **checkpoint**> onto stable storage.
3. During recovery we need to consider only the most recent transaction Ti that started before the checkpoint, and transactions that started after *Ti*.
   a. Scan backwards from end of log to find the most recent <**checkpoint**> record
   b. Continue scanning backwards till a record <*Ti* **start**> is found.
   c. Need only consider the part of log following above **star**t record. Earlier part of log can be ignored during recovery, and can be erased whenever desired.
   d. For all transactions (starting from *Ti* or later) with no <*Ti* **commit**>, execute **undo***(Ti)*. (Done only in case of immediate modification.)
   e. Scanning forward in the log, for all transactions starting from *Ti* or later with a <*Ti* **commit**>, execute **redo***(Ti)*.

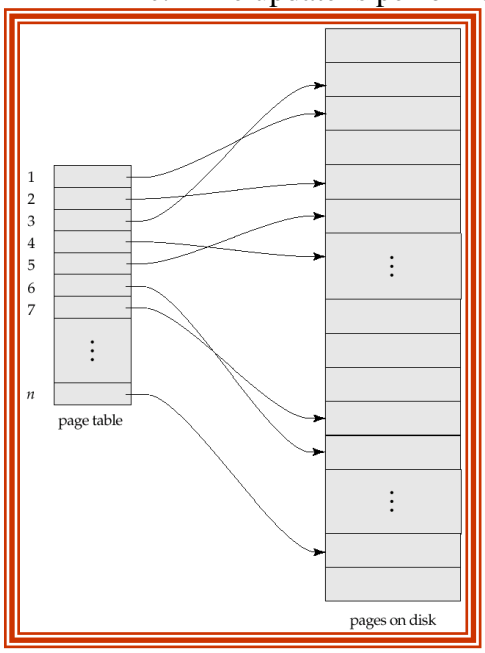
Example of Checkpoints

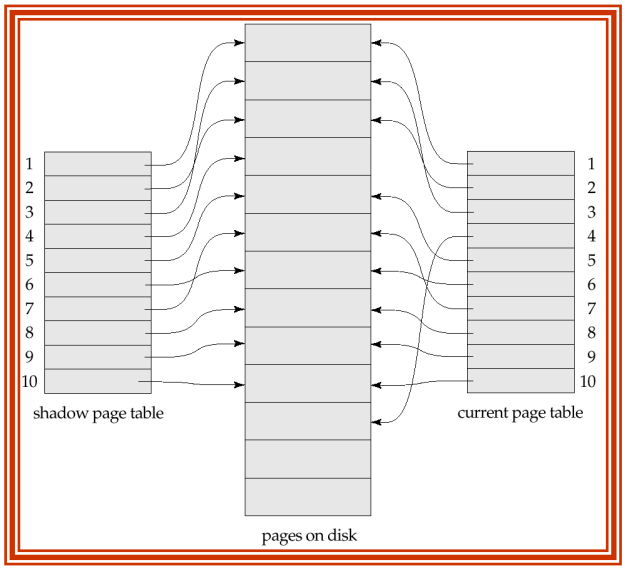


1. *T*1 can be ignored (updates already output to disk due to checkpoint)
2. *T*2 and *T*3 redone.
3. *T*4 undone

Shadow Paging

1. **Shadow paging** is an alternative to log-based recovery; this scheme is useful if transactions execute serially
2. Idea: maintain *two* page tables during the lifetime of a transaction –the **current page table**, and the **shadow page table**
3. Store the shadow page table in nonvolatile storage, such that state of the database prior to transaction execution may be recovered.
    a. Shadow page table is never modified during execution
4. To start with, both the page tables are identical. Only current page table is used for data item accesses during execution of the transaction.
5. Whenever any page is about to be written for the first time
    a. A copy of this page is made onto an unused page.
    b. The current page table is then made to point to the copy
    c. The update is performed on the copy





1. To commit a transaction :
    a) Flush all modified pages in main memory to disk
    b) Output current page table to disk
    c) Make the current page table the new shadow page table, as follows:
        ★ keep a pointer to the shadow page table at a fixed (known) location on disk.
        ★ to make the current page table the new shadow page table, simply update the pointer to point to current page table on disk
2. Once pointer to shadow page table has been written, transaction is committed.
3. No recovery is needed after a crash — new transactions can start right away, using the shadow page table.
4. Pages not pointed to from current/shadow page table should be freed (garbage collected).
5. Advantages of shadow-paging over log-based schemes

a) no overhead of writing log records
b) recovery is trivial

6. Disadvantages :
    a) Copying the entire page table is very expensive
        ⇨ Can be reduced by using a page table structured like a B+-tree
            - No need to copy entire tree, only need to copy paths in the tree that lead to updated leaf nodes
    b) Commit overhead is high even with above extension
        ⇨ Need to flush every updated page, and page table
    c) Data gets fragmented (related pages get separated on disk)
    d) After every transaction completion, the database pages containing old versions of modified data need to be garbage collected
    e) Hard to extend algorithm to allow transactions to run concurrently
        ⇨ Easier to extend log based schemes