

Database systems

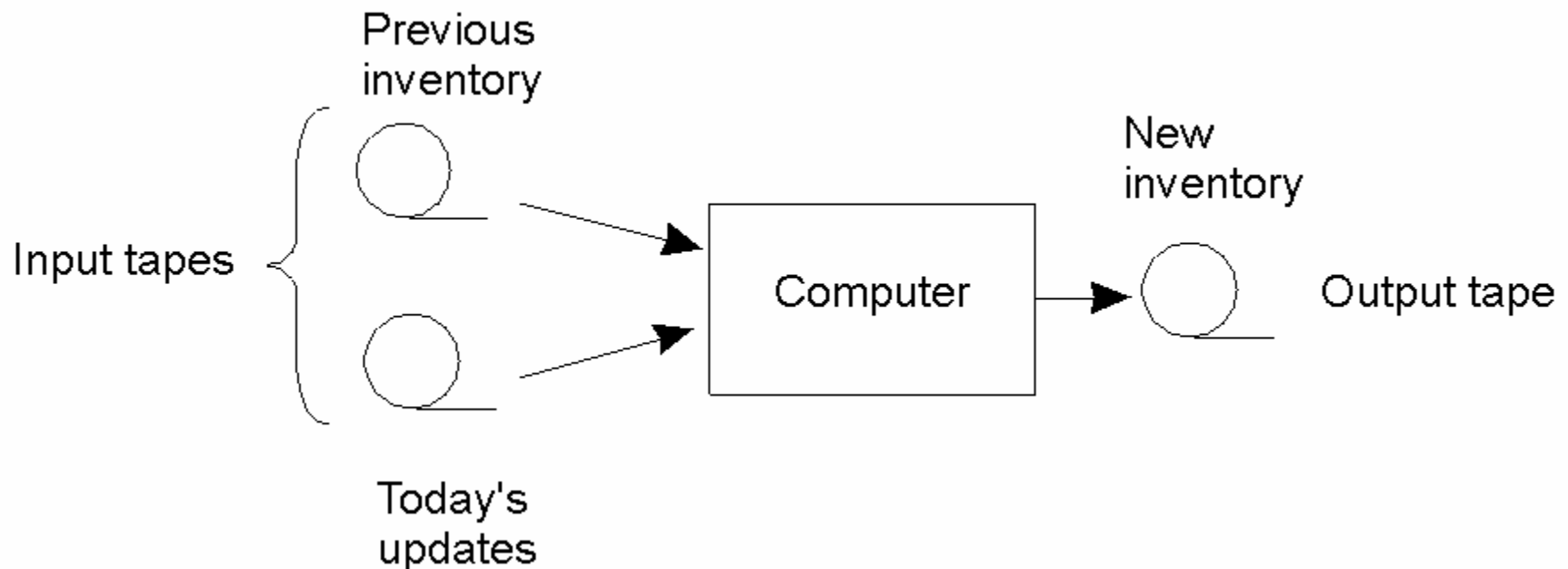
- Database: a collection of shared data objects (d_1, d_2, \dots, d_n) that can be accessed by users
 - every database has some correctness constraints defined on it (called *consistency assertions* or *integrity constraint*)
 - a database is said to be *consistent* if the values of its data satisfy these constraints
- A user performs interacts with a database through complex operations called *transactions*
 - a transaction consists of a sequence of read, write, compute statements that refers to data objects in the database
 - examples: on-line booking, bank teller operations, ...

Transactions

- Transactions: set of actions on a database that are grouped in a single logical unit of interaction
 - nomenclature:
 - *query*: read-only transaction
 - *update*: transaction modifies at least one object
 - assumptions:
 - transactions preserve consistency
 - transactions terminate in finite time
 - ACID properties
 - Atomic
 - Consistent
 - Isolated
 - Durable

The Transaction Model (1)

- Old method of updating a master tape is fault tolerant.
 - Contrast with modern online database that is updated in place



The Transaction Model (2)

- The atomicity of transactions can be recreated with special primitives.

Primitive	Description
BEGIN_TRANSACTION	Make the start of a transaction
END_TRANSACTION	Terminate the transaction and try to commit
ABORT_TRANSACTION	Kill the transaction and restore the old values
READ	Read data from a file, a table, or otherwise
WRITE	Write data to a file, a table, or otherwise

The Transaction Model (3)

- Example
 - a) Transaction to reserve three flights commits
 - b) Transaction aborts when third flight is unavailable

```
BEGIN_TRANSACTION
reserve WP -> JFK;
reserve JFK -> Nairobi;
reserve Nairobi -> Malindi;
END_TRANSACTION
```

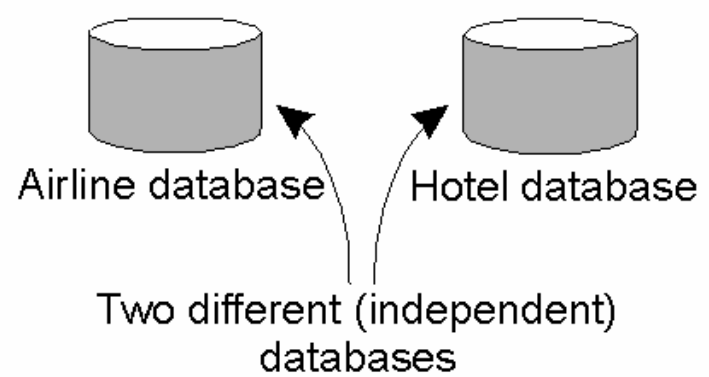
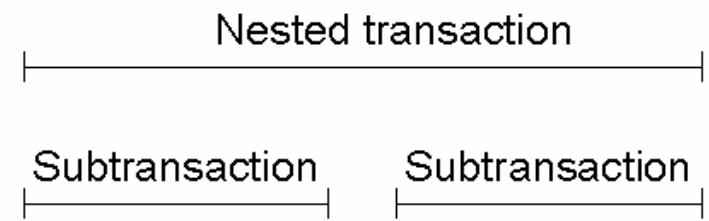
(a)

```
BEGIN_TRANSACTION
reserve WP -> JFK;
reserve JFK -> Nairobi;
reserve Nairobi -> Malindi full =>
ABORT_TRANSACTION
```

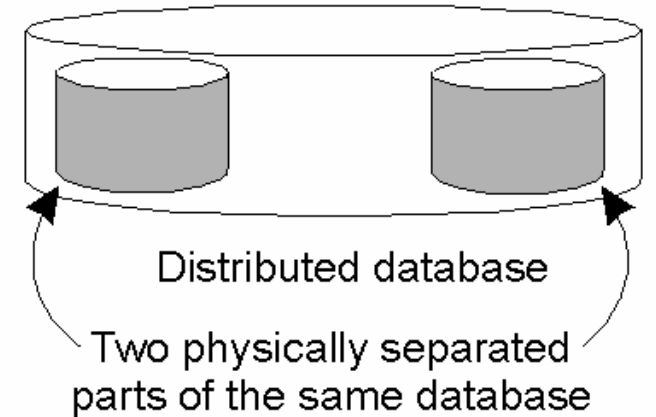
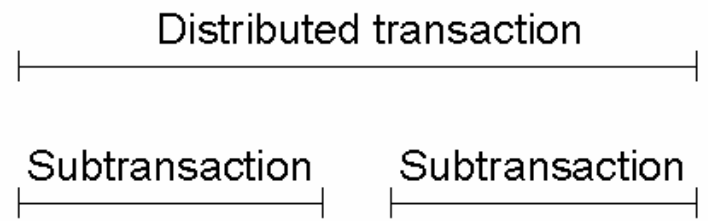
(b)

Nested and Distributed Transactions

- Difference between nested and distributed transaction
 - a) A nested transaction occurs on logically and physically separate databases (independent sub-transactions)
 - b) A distributed transaction takes place on a logically single but physically distributed database



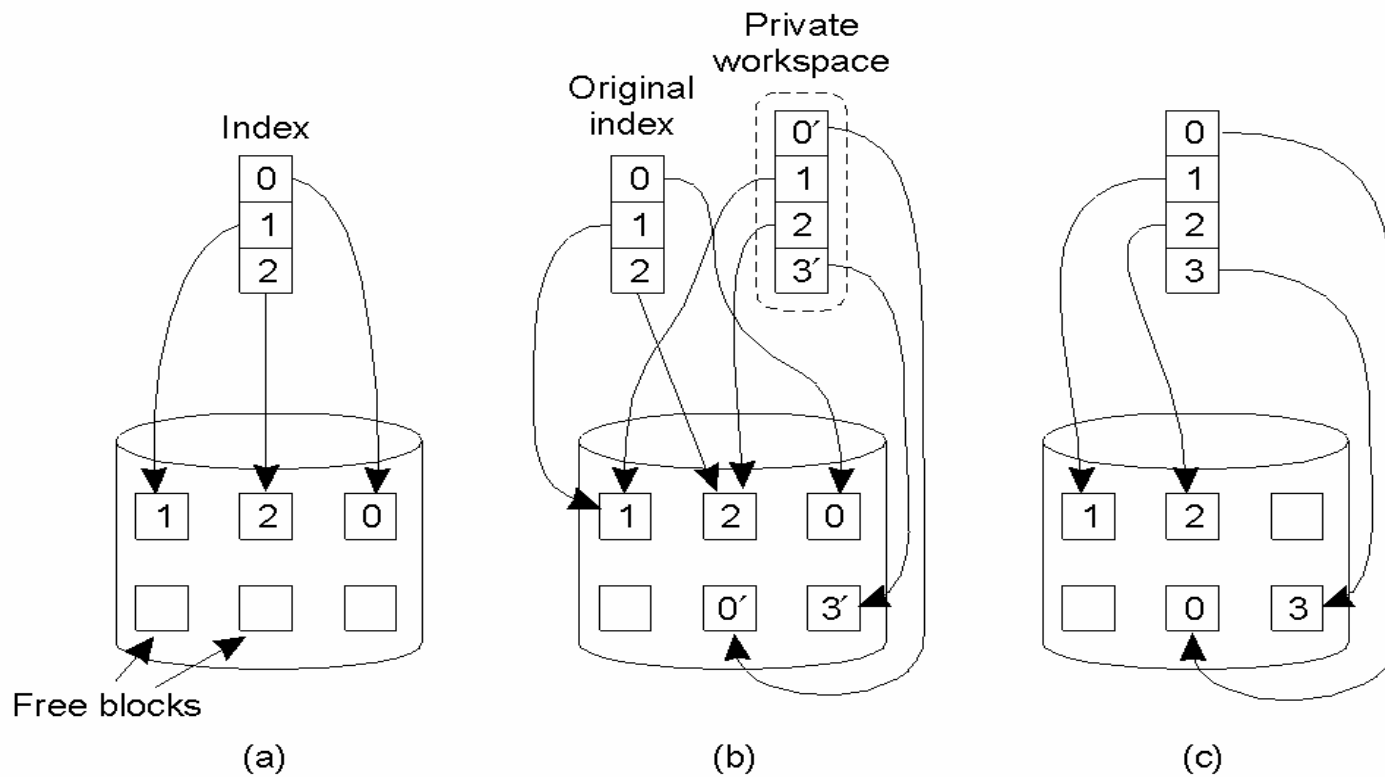
(a)



(b)

Implementation #1: Private Workspace

- a) The file index and disk blocks for a three-block file
- b) The situation after a transaction has modified block 0 and appended block 3
- c) After committing



Implementation #2: Writeahead Log

- a) A transaction
- b) – d) The log before each statement is executed

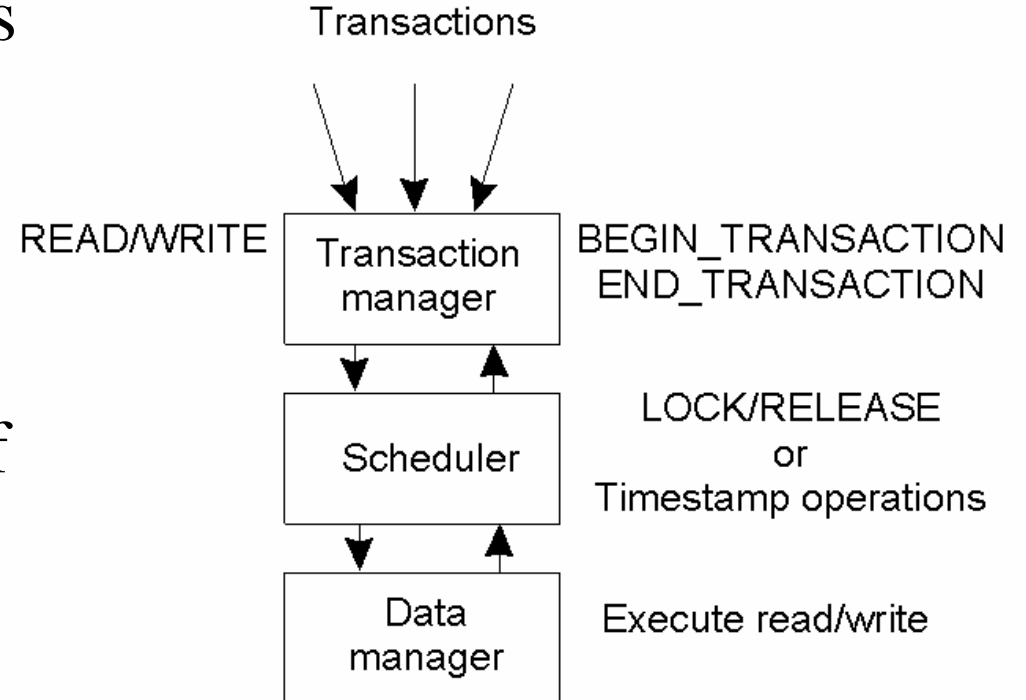
x = 0;	Log	Log	Log
y = 0;			
BEGIN_TRANSACTION;			
x = x + 1;	[x = 0 / 1]	[x = 0 / 1]	[x = 0 / 1]
y = y + 2		[y = 0/2]	[y = 0/2]
x = y * y;			[x = 1/4]
END_TRANSACTION;			
(a)	(b)	(c)	(d)

Concurrency control (1)

- *Concurrency control* is the set of mechanisms put in place to preserve consistency and isolation
 - If we were to execute only one transaction at a time the (i.e. *sequentially*) implementation of consistency and isolation would be trivially solved
 - However most of the time we are interested in providing concurrency – allowing several transactions to be executed simultaneously
- The implementation of distributed transactions further complicates concurrency control because requires handling multiple databases

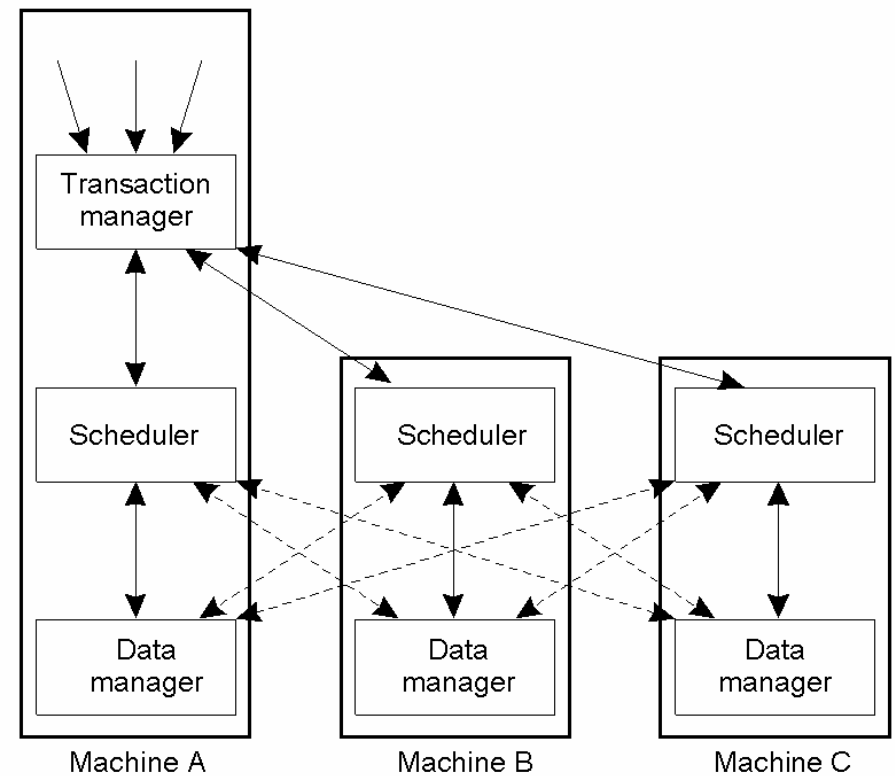
Concurrency Control (2)

- Concurrency control is best understood in terms of three managers
 - Data manager
 - Scheduler
 - Transaction manager
- The three managers implement a division of tasks
 - concurrency control is responsibility of the scheduler



Concurrency Control (3)

- The managers scheme can be extended to the distributed scheme



Serializability

BEGIN_TRANSACTION
 $x = 0;$
 $x = x + 1;$
 END_TRANSACTION

(a)

BEGIN_TRANSACTION
 $x = 0;$
 $x = x + 2;$
 END_TRANSACTION

(b)

BEGIN_TRANSACTION
 $x = 0;$
 $x = x + 3;$
 END_TRANSACTION

(c)

Schedule 1	$x = 0;$ $x = x + 1;$ $x = 0;$ $x = x + 2;$ $x = 0;$ $x = x + 3$	Legal
Schedule 2	$x = 0;$ $x = 0;$ $x = x + 1;$ $x = x + 2;$ $x = 0;$ $x = x + 3;$	Legal
Schedule 3	$x = 0;$ $x = 0;$ $x = x + 1;$ $x = 0;$ $x = x + 2;$ $x = x + 3;$	Illegal

(d)

- a) – c) Three transactions T_1 , T_2 , and T_3
- d) Possible schedules

Serializability

BEGIN_TRANSACTION
 $x = 0;$
 $x = x + 1;$
 END_TRANSACTION

(a)

BEGIN_TRANSACTION
 $x = 0;$
 $x = x + 2;$
 END_TRANSACTION

(b)

BEGIN_TRANSACTION
 $x = 0;$
 $x = x + 3;$
 END_TRANSACTION

(c)

Schedule 1	$x = 0;$ $x = x + 1;$ $x = 0;$ $x = x + 2;$ $x = 0;$ $x = x + 3$	Legal
Schedule 2	$x = 0;$ $x = 0;$ $x = x + 1;$ $x = x + 2;$ $x = 0;$ $x = x + 3;$	Legal
Schedule 3	$x = 0;$ $x = 0;$ $x = x + 1;$ $x = 0;$ $x = x + 2;$ $x = x + 3;$	Illegal

(d)

- a) – c) Three transactions T_1 , T_2 , and T_3
- d) Possible schedules

Concurrency Control Theory

- We want to be able to execute several transactions simultaneously
 - serial execution preserves consistency (because each transaction individually preserves consistency) but is inefficient
 - what we really want is concurrent (i.e. interleaved) execution
- What we need is a method that enables us to answer the following questions:
 - Give the log of a certain execution of a set of transaction, is that execution legal? (i.e. preserves consistency and isolation)

Conflicts

- Two transactions T_1 and T_2 are said to conflict if
 - they access the same data objects, and
 - at least one access is a write, i.e. r-w, w-r, or w-w conflict
- Note that if two transactions have no conflict, then their execution can be interleaved without restrictions

Logs

- *Log*: chronological order of actions performed by transactions under a concurrency control algorithm
 - a log over T describes an interleaved execution of T_0, T_1, \dots, T_n
 - Example:

$T_1 = r1[x] \ r1[z] \ w1[x]$

$T_2 = r2[y] \ r2[z] \ w2[y]$

$T_3 = w3[x] \ r3[y] \ w3[z]$

L1

w3[x] r1[x] r3[y] r2[y] w3[z] r2[z] r1[z] w2[y] w1[x]

L2

w3[x] r3[y] w3[z] r2[y] r2[z] w2[y] r1[x] r1[z] w1[x]

Equivalent logs

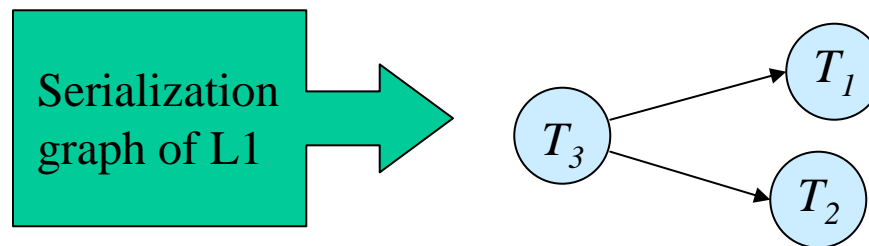
- Two logs are *equivalent* if
 - all the transactions in both logs see the same state of the database,
 - they leave the database in the same final state
- Two logs over a transaction set are clearly equivalent iff:
 - every read operation reads from (i.e. sees the value written by) the same write in both logs, and
 - both logs have the same final writes
- Example: L1 and L2 in previous example are equivalent

Serial Logs

- *Serial* log:
 - a log in which all actions of a transaction terminate before any action of the next transaction starts
 - example: L2 is a serial log
- *Serializable* log:
 - a log in which actions from several transactions T_0, T_1, \dots, T_n are interleaved, and that has the same output and the same effect on the database as the serial execution of a permutation of T_0, T_1, \dots, T_n
 - Example: log L1 is equivalent to serial log L2
- A serializable log is equivalent to a serial log, thus represents a correct execution
 - question: is there some criterion for telling if a log is serializable?

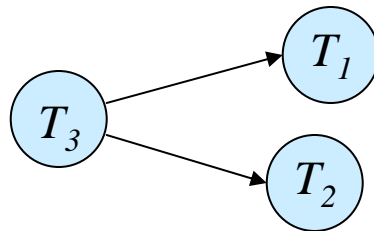
Serialization Graph

- Let L be a log over a set of transactions T_0, T_1, \dots, T_n
 - the *serialization graph* $SG(L)$ of L is a directed graph in which the nodes are the transactions T_i , and whose edges satisfy the condition:
 - edge $T_i \rightarrow T_j$ then for some x either $ri[x] < wj[x]$ or $wi[x] < rj[x]$ or $wi[x] < wj[x]$
 - i.e. an edge $T_i \rightarrow T_j$ denotes a conflict between actions of C and T_j
 - example:



The Serializability Theorem

- Th.: A log L is serializable iff $SG(L)$ is acyclic
 - no cycles in $AG(L) \Leftrightarrow L$ is serializable
- The serial log corresponding to L can be determined by topologically sorting $AG(L)$
 - example:



$T_3 \rightarrow T_1 \rightarrow T_2$ OR $T_3 \rightarrow T_2 \rightarrow T_1$