Algorithm comparisons

- Ricart-Agrawala's can be seen as an optimization of Lamport's:
  - RELEASE messages are merged with REPLY messages
- Basic differences:
  - Lamport's idea is to maintain (partially) coherent copies of a replicated data structure - the *request_queue*
  - Ricart-Agrawala does away with the data structure and just propagates state changes
  - messages needed for CS execution in the two schemes:
    - 3(N-1) vs. 2(N-1)
- Do we need FIFO assumption on Ricart-Agrawala's algorithm? Why?

Maekawa's Algorithm

- Difference with respect to previous algorithms:
  - a site does not request permission from every other site but only from a subset - called *request set*
- The request sets of any two sites have at least one site in common:

$$\forall i \; \forall j : 1 \leq i,j \leq N :: R_i \cap R_j \neq \emptyset$$

- The basic idea is that each pair of sites is going to have a third site mediating conflicts between the pair

Maekawa's algorithm steps

- Requesting the CS
  - S_i sends a message **REQUEST(*i*)** to all the sites in R_i
  - when S_j receives a request from S_i, it returns a **REPLY** to S_i if it has not sent a **REPLY** since receiving the latest **RELEASE** message. Otherwise the request is enqueued.
- Executing the CS
  - A site S_i executes the CS only after receiving **REPLY** messages from all the sites in R_i
- Releasing the CS
  - When done, a site S_i sends a **RELEASE** message to all the sites in R_i
  - When a site receives a RELEASE message, it sends a REPLY message to the next site waiting in the queue and removes it

Construction of the request set

- The request sets are constructed to satisfy the following conditions:
  - $\forall i\ \forall j : 1 \leq i,j \leq N :: R_i \cap R_j \neq \emptyset$
    - necessary for correctness
  - $\forall i : 1 \leq i \leq N :: S_i \in R_i$
    - necessary for correctness (note: this condition, like the need for FIFO comm., is really needed only in the extended version of the algorithm)
  - $\forall i : 1 \leq i \leq N :: |R_i| = K$
    - all R_i have equal size, so all sites do equal work to access the CS
  - Any site S is contained in K of the R_i's
    - the same number of sites are requesting permission from each site (no bottleneck)

More on the request set

- All the previous conditions are satisfied if N can be expressed as:

$$N = K(K - 1) + 1$$

(examples: N = 3 and K = 2, N = 7 and K = 3, etc.)

  - Note that, for large N, K $\approx \sqrt{N}$

- Otherwise one of the last two conditions must be relaxed
  - for example, $| R_i | = K$ no longer true for all i

Notes on Maekawa's algorithm

- Performance:
  - $3\sqrt{N}$ messages are needed for execution of the CS
  - synchronization delay is $2T$
- Problem: the algorithm is deadlock prone!
  - there is a variant of the algorithm that can prevent the deadlock by using a priority-based preempting scheme
  - this variant requires additional messages (up to $5\sqrt{N}$)

Election Problem

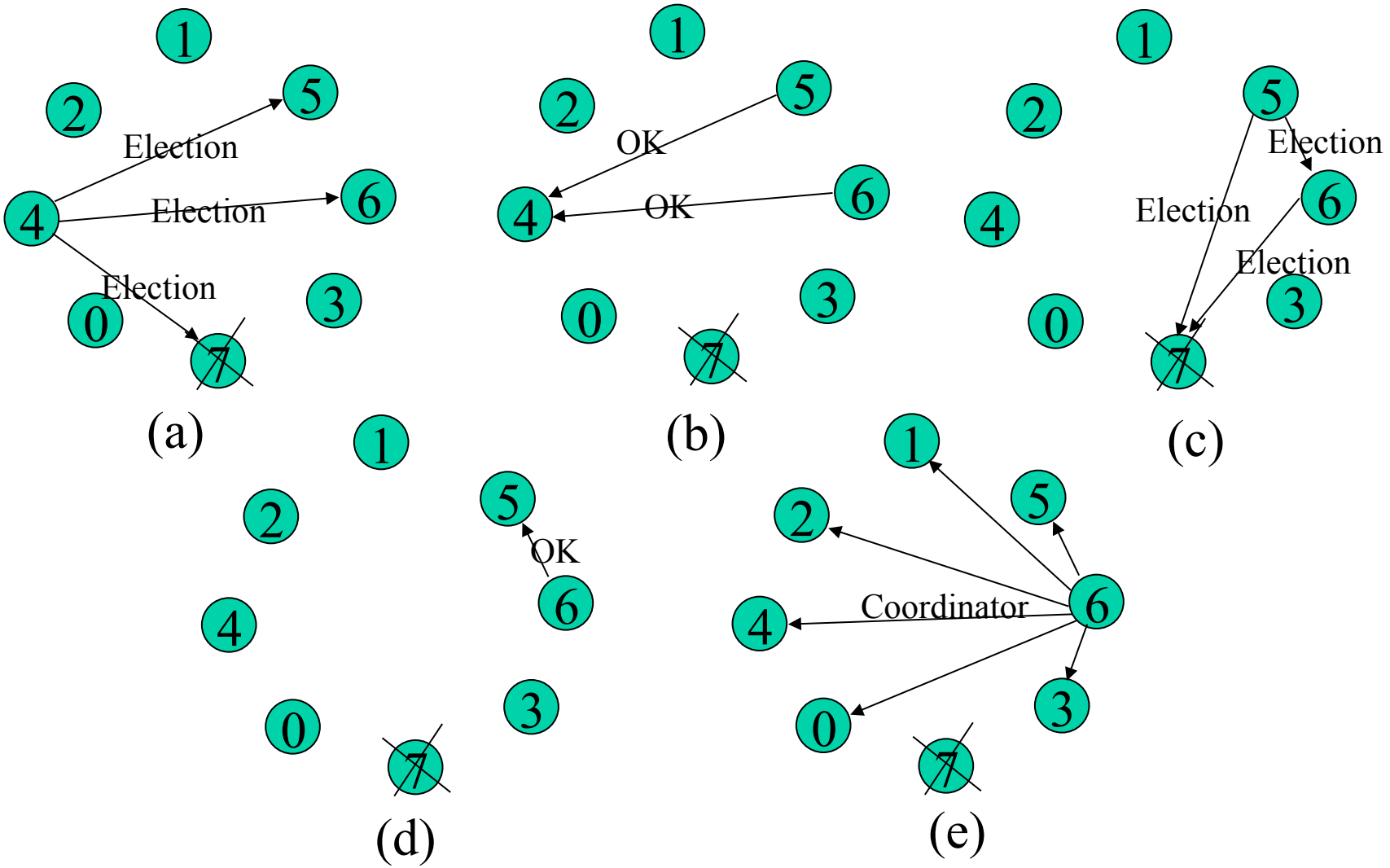
- Many distributed algorithms require one process to act as coordinator.
- How do they elect the new coordinator initially or when the current coordinator is down?
- Assume every process knows the process number of every other processes
- No knowledge about nodes status (up or down)
- Goal:
  - When an election starts, it concludes with all processes agreeing on who the new coordinator is to be.

The Bully Algorithm (Garcia-Molina)

- *P* sends an *ELECTION* message to all processes with higher numbers
- If no one responds, *P* wins the election and becomes a coordinator
- If one of the higher-ups answers, it takes over. *P*'s job is done.

- Repeat the above steps until one wins
- The new coordinator sends *Coordinator* message to all other processes

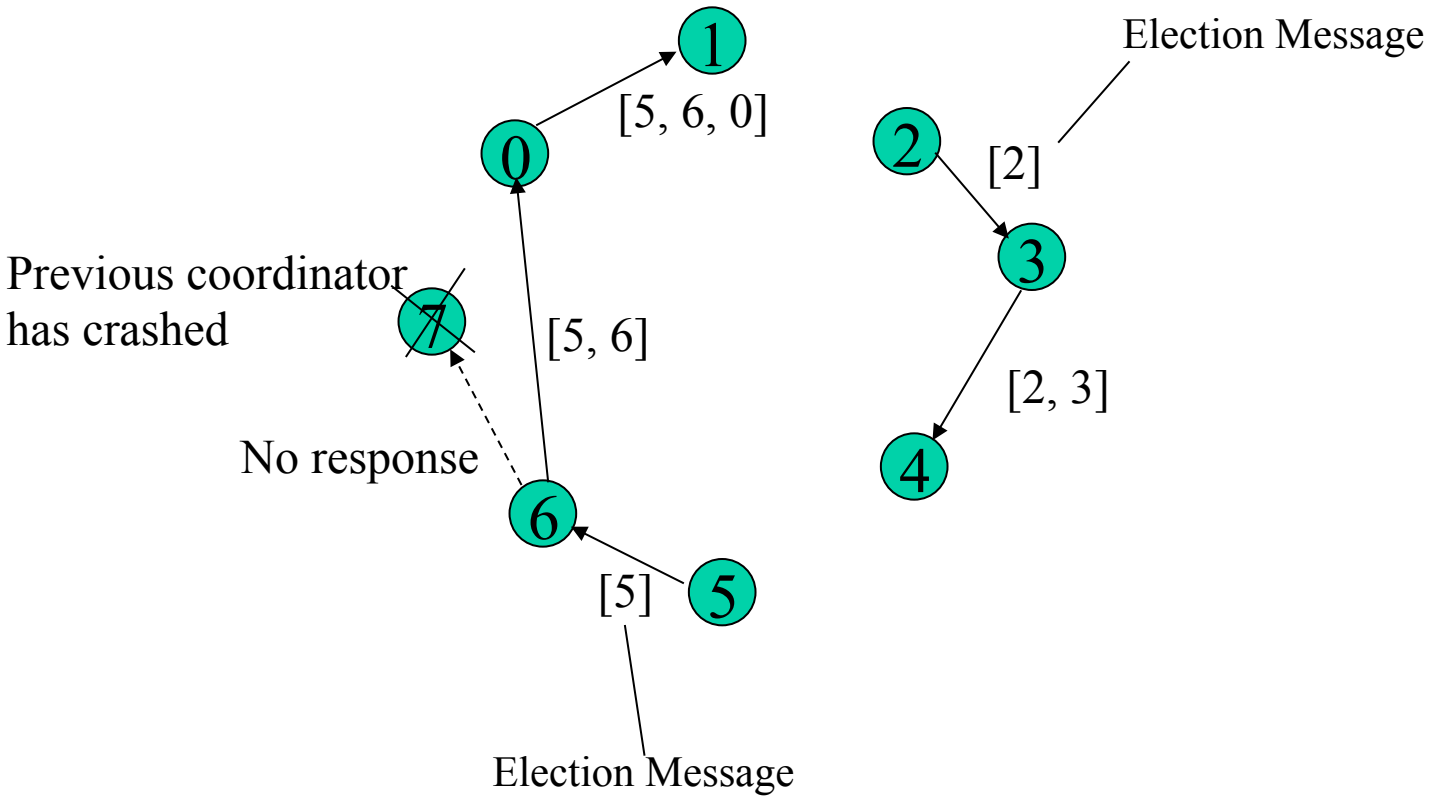
An Example


1
2
5
Election
4
Election
6
Election
0
3
7
(a)
OK
OK
(b)
Election
Election
(c)
OK
(d)
Coordinator
(e)

A Ring Algorithm

- Each process knows who its successor is.
- Any process that notices the coordinator is not functioning builds an *ELECTION* message containing its own process number and sends the message to its successor
- At each step along the way, the sender adds its own process number to the list
- The process with the highest number wins the election
- When the message gets back, the first process changes the message type to *COORDINATOR* and circulates the message once again.

An Example


1
[5, 6, 0]
0
Election Message
2
[2]
3
Previous coordinator
has crashed
7
[5, 6]
[2, 3]
No response
4
6
[5]
5
Election Message

Database systems

- Database: a collection of shared data objects (d1, d2, … dn) 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 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 (all or nothing)
    - Consistent (satisfy consistent rules)
    - Isolated (not interfere with each other)
    - Durable (committed transaction will not be lost)

The Transaction Model (1)

- The atomicity of transactions can be re-created 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 (2)

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

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

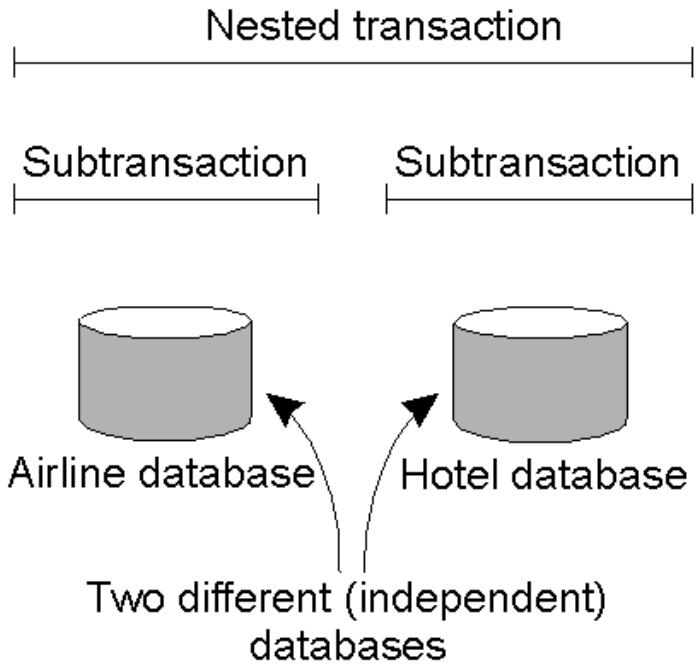
(a)

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

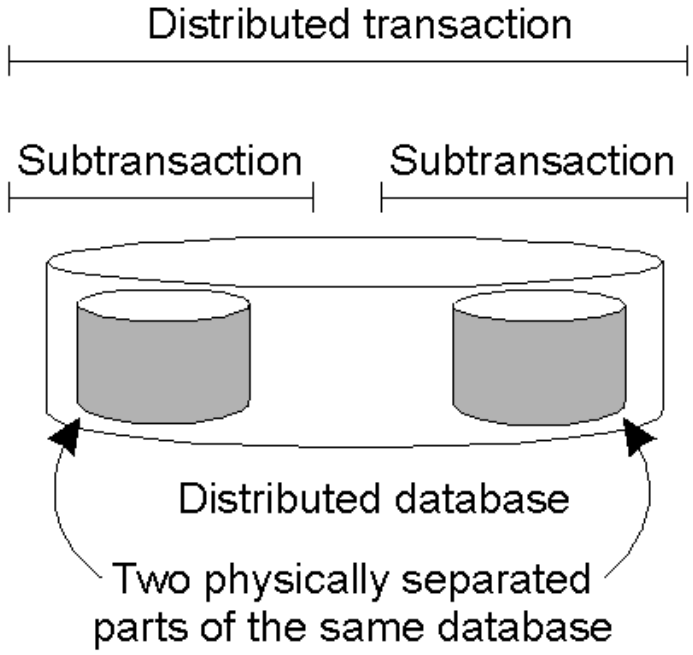
(b)

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



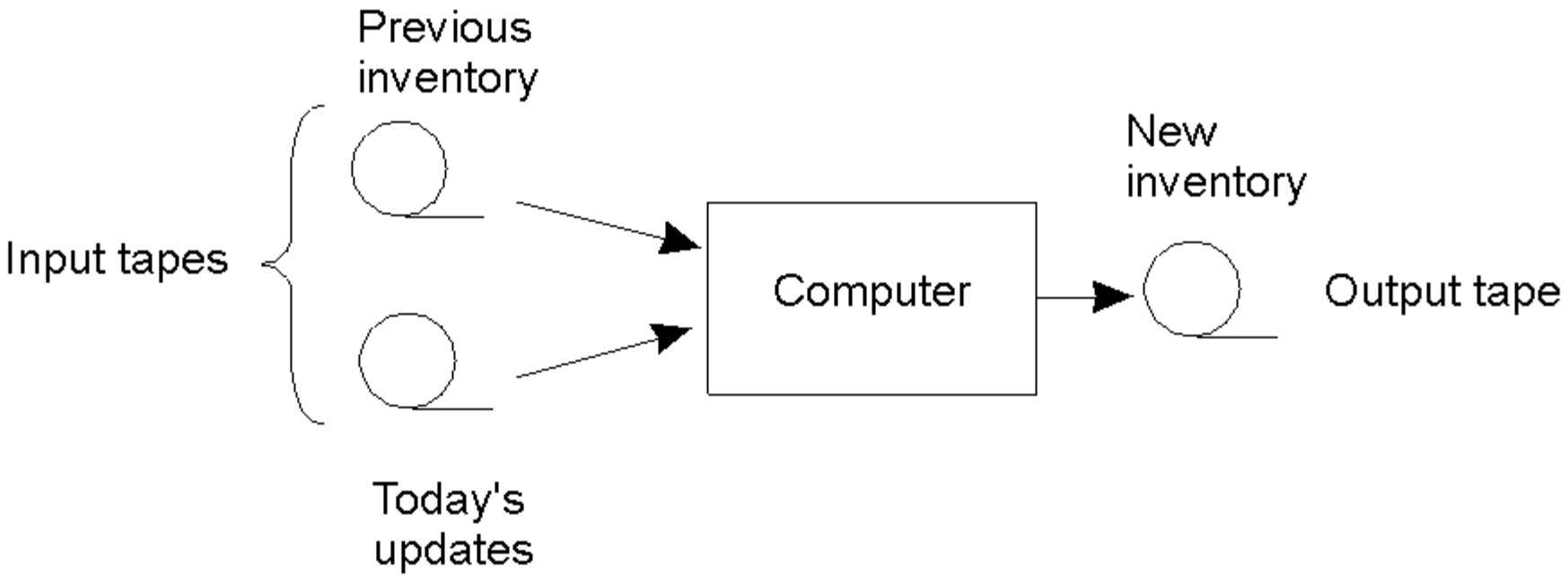
(a)



(b)

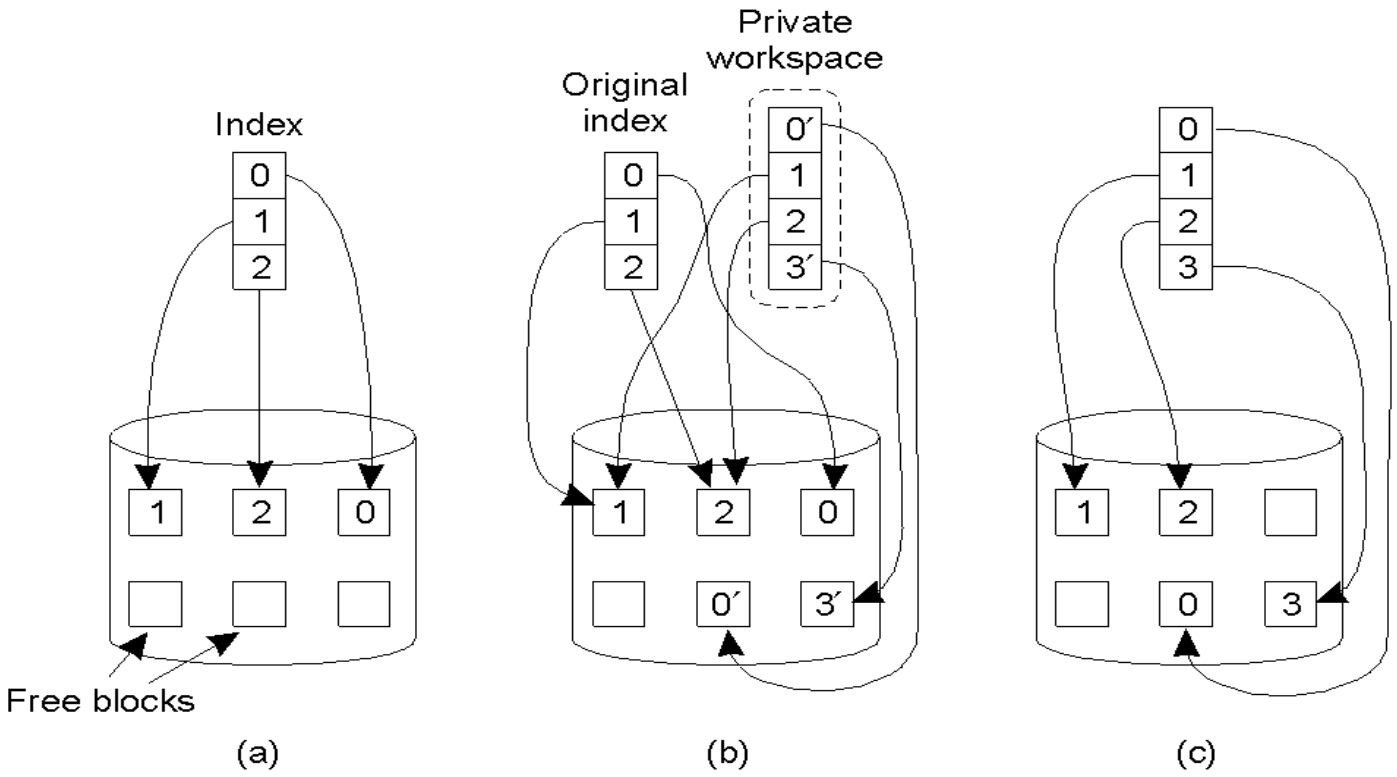
Traditional Transaction Updates

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



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)