Storing Data: Disks and Files, File Organizations and Indexes, Tree-Structured Indexes, Hash-Based Indexes, Transaction Management

Wednesday, 6 May 2020 7:21 PM

Disks and Files
11.1 Memory Hierarchy
• Primary Storage: main memory.
fast access, expensive.
• Secondary storage: hard disk.
slower access, less expensive.
• Tertiary storage: tapes, cd, etc.
slowest access, cheapest.

11.2 Disks
Characteristics of disks:
• collection of platters
• each platter = set of tracks
• each track = sequence of sectors (blocks)
• transfer unit: 1 block (e.g. 512B, 1KB)
• access time depends on proximity of heads to required block
access
• access via block address (p, t, s)
• Data must be in memory for the DBMS to operate on it.
• If a single record in a block is needed, the entire block is transferred.

Access time includes:
• seek time (find the right track, e.g. 10msec)
• rotational delay (find the right sector, e.g. 5msec)
• transfer time (read/write block, e.g. 10µsec)
Random access is dominated by seek time and rotational delay
1. Improving Disk Access:
Use knowledge of data access patterns.
2. Keeping Track of Free Blocks
– Maintain a list of free blocks.
– Use bitmap.
3. Using OS File System to Manage Disk Space
– extend OS facilities, but
– not rely on the OS file system.
(portability and scalability)

Buffer Pool
• The request_block operation replaces read block in all file access algorithms.
• If block is already in buffer pool:
– no need to read it again
– use the copy there (unless write-locked)
• If block is not already in buffer pool:
– need to read from hard disk into a free frame
– if no free frames, need to remove block using a buffer replacement policy.
• The release_block function indicates that block is no longer in use ⇒good
candidate for removal.

Buffer Replacement Policies
Several schemes are commonly in use:
• Least Recently Used (LRU)

– release the frame that has not been used for the longest period.
– intuitively appealing idea but can perform badly
• First in First Out (FIFO)
– need to maintain a queue of frames
– enter tail of queue when read in
• Most Recently Used (MRU): release the frame used most recently
• Random
No is guaranteed better than the other.
For DBMS, we may predict accesses better.

Example1:

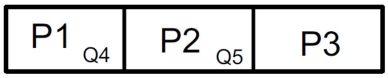
Data pages: P1, P2, P3, P4

Q1: read P1; Q2: read P2;

Q3: read P3; Q4: read P1;

Q5: read P2; Q6: read P4;

Buffer:



Regarding Q6,

- LRU: Replace P3
- MRU: Replace P2
- FIFO: Replace P1
- Random: randomly choose one buffer to replace

Example 2:
Data pages: P1, P2, …, P11
10 buffer pages as in Example 1
Q1: read P1, P2,…, P11;
Q2, read P1, P2,…, P11;
Q3: Read P1, P2,…,P11

LRU/FIFO: I/O P1, P2, …, P11 for each query.

MRU performs the best.

File Organizations and Indexes
COST MODEL
D: average time to read or write a disk page.
C : average time to process a record.
H : the time required to apply a hash function to a record.

3 File Organizations:
Heap Files.
Sorted Files.
Hashed Files.

Scan:
B(D +RC) where
• B is the number of pages, and
• R is the average number of records in a page (block).
SWES:
• 0.5B(D +RC) on average if the selection is specified on a key.
• Otherwise B(D +RC).

Summary

File Type	Scan	Equality Search	Range Search	Insert	Delete
Heap	BD	0.5 BD	BD	Search + D	Search + D
Sorted	B D	D log B	D log B + # matches	Search + BD	Search + BD
Hashed	1.25 BD	D	1.25 BD	2 D	Search + BD

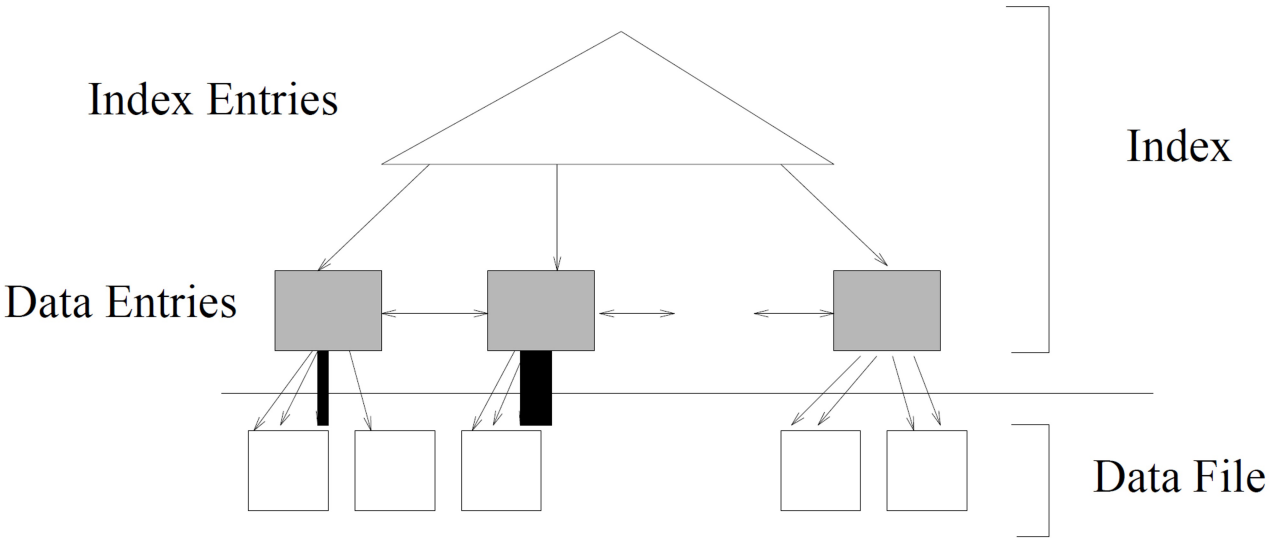
A Comparison of I/O Costs

Indexes

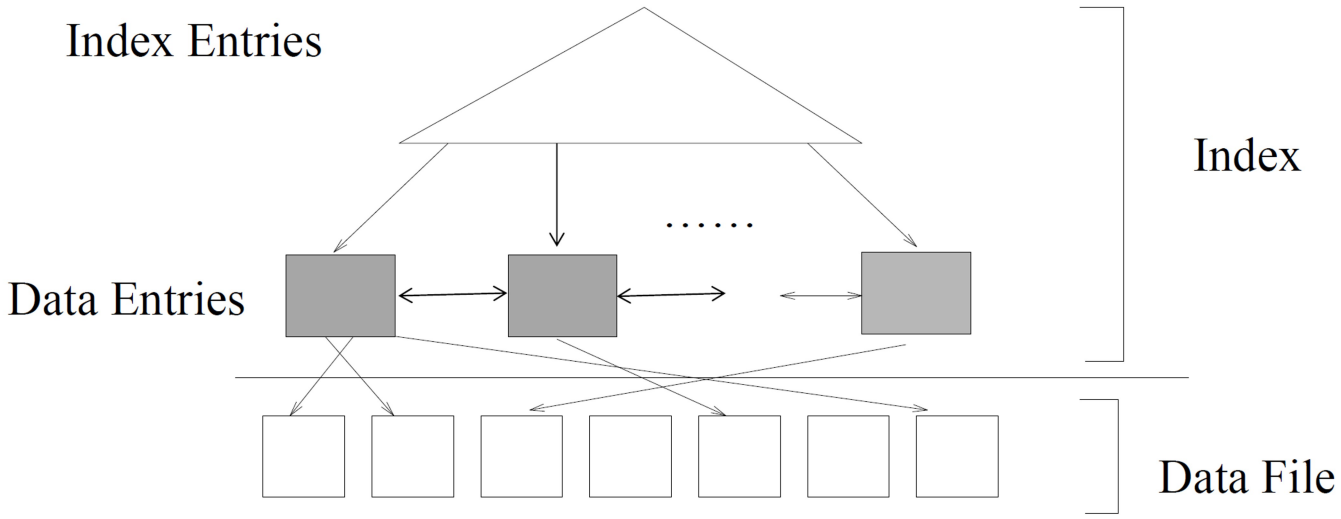
Indexing Structure
Index is collection of data entries k∗.
Each data entry k∗ contains enough information to retrieve (one or more) records with search key value k.

Clustered Index

- Clustered: a file is organized of data records is the same as or close to the ordering of data entries in some index.
- Typically, the search key of file is the same as the search key of index.

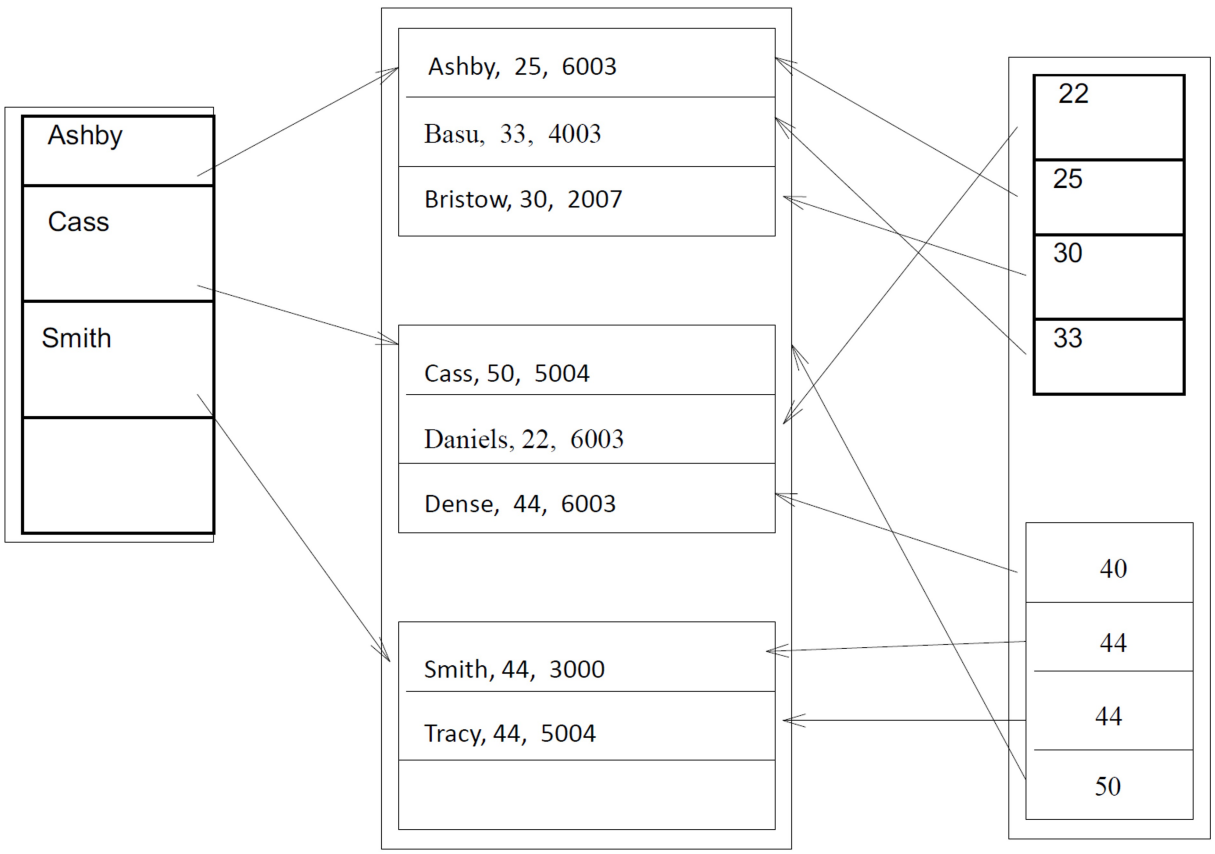


Unclustered Index



- Clustered indexes are relatively expensive to maintain.
- A data file can be clustered on at most one search key.

Dense VS Sparse Indexes
§ Dense: it contains (at least) one data entry
for every search key value.
§ Sparse: otherwise.

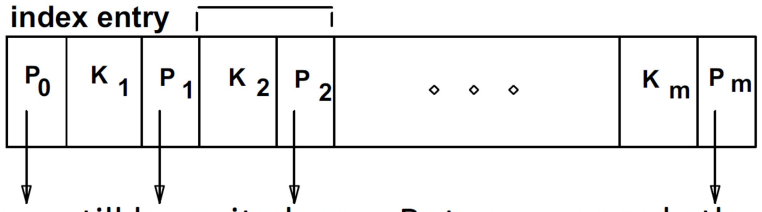


Sparse Index VS Dense Index

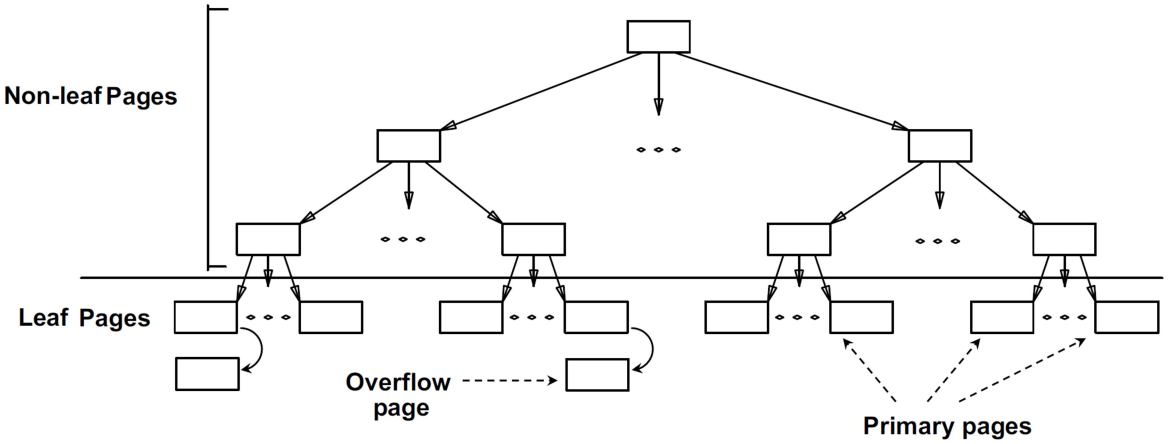
Primary and Secondary Indexes
§ Primary: Indexing fields include primary key.
§ Secondary: otherwise.
There may be at most one primary index for file.
Composite search keys: search key contains several fields.

Tree-Structured Indexes
Introduction
As for any index, 3 alternatives for data entries k*:
– Data record with key value k
– <k, rid of data record with search key value k>
– <k, list of rids of data records with search key k>
§ Choice is orthogonal to the indexing technique used to locate data entries k*.
§ Tree-structured indexing techniques support both range searches and equality searches.
§ ISAM: static structure; B+ tree: dynamic, adjusts gracefully under inserts and deletes.

ISAM (Indexed Sequential Access Method)

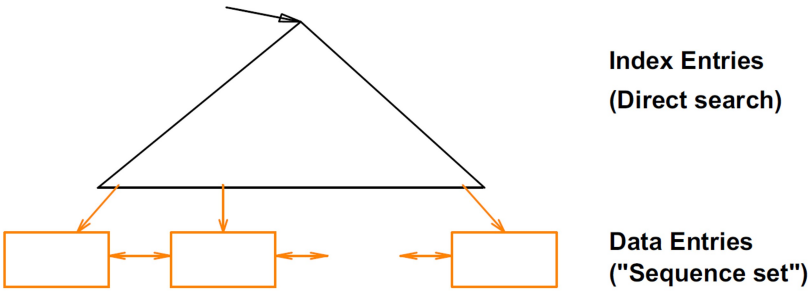


Index file may still be quite large. But we can apply the idea repeatedly!



B+ Tree: Most Widely Used Index

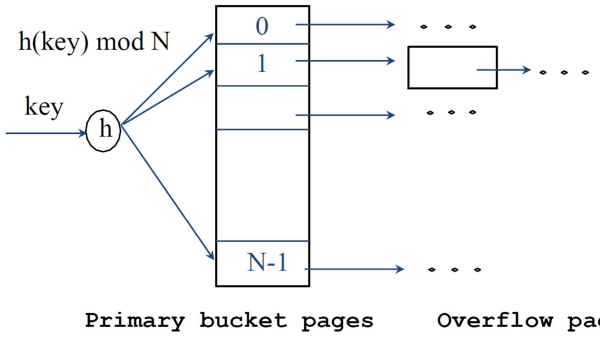
- Insert/delete at log $_F$ N cost; keep tree *height-balanced*. (F = fanout, N = # leaf pages)
- Minimum 50% occupancy (except for root). Each node contains **d** <= *m* <= 2**d** entries.
- The parameter **d** is called the *order* of the tree.
- Supports equality and range-searches efficiently.



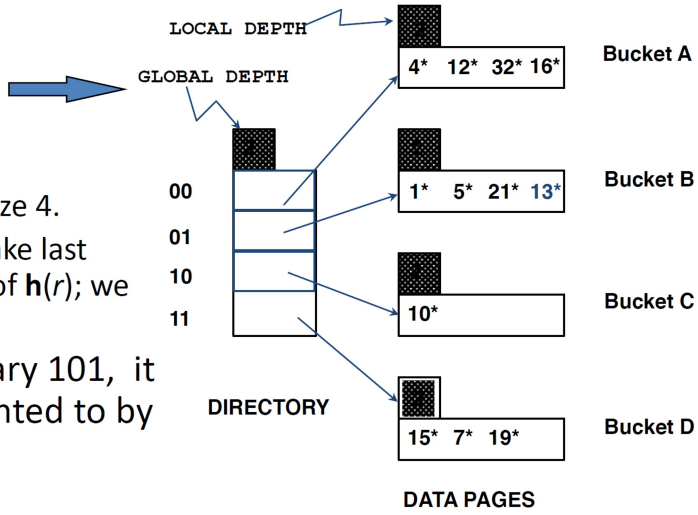
Hash-Based Indexes
Introduction
As for any index, 3 alternatives for data entries k*:
1. Data record with key value k
2. <k, rid of data record with search key value k>
3. <k, list of rids of data records with search key k>
§ Choice orthogonal to the indexing technique
§ Hash-based indexes are the best for equality selections. Cannot support range searches.
§ Static and dynamic hashing techniques exist; trade-offs similar to ISAM vs. B+ trees.

Static Hashing

- # primary pages (index data entry pages) fixed, allocated sequentially, never de-allocated; overflow pages if needed.
- h(*k*) mod M = bucket to which data entry with key *k* belongs. (M = # of buckets)



Example



- Directory is array of size 4.
- To find bucket for *r*, take last `*global depth*' # bits of **h**(*r*); we denote *r* by **h**(*r*).
  - If **h**(*r*) = 5 = binary 101, it is in bucket pointed to by 01.

❖ **Insert**: If bucket is full, *split* it (*allocate new page, re-distribute*).

❖ *If necessary*, double the directory. (As we will see, splitting a bucket does not always require doubling; we can tell by comparing *global depth* with *local depth* for the split bucket.)

Transaction Management
Transactions, Recovery and Concurrency (I)

Transaction Processing
Three kinds of operations may be used in a transaction:
– Read.
– Write.
– Computation.

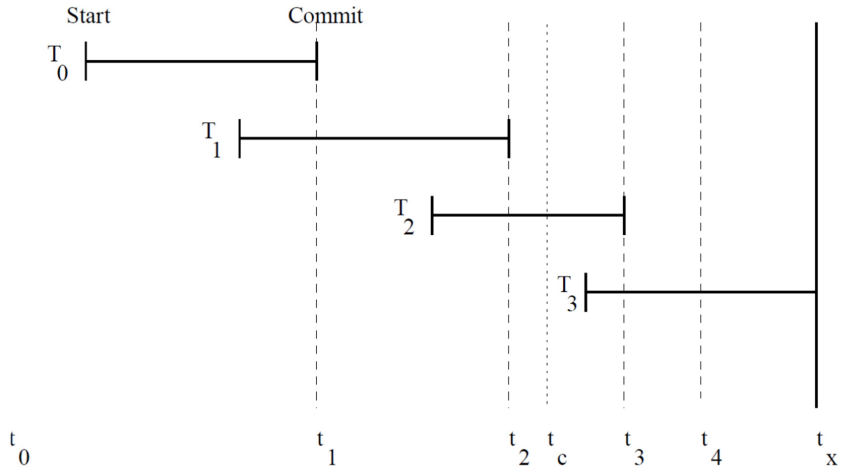
Desirable Properties of Transaction Processing **ACID**

- *Atomicity:* A transaction is either performed in its entirety or not performed at all.

- *Consistency preservation:* A correct execution of the transaction must take the database from one consistent state to another.

- *Isolation:* A transaction should not make its updates visible to other transactions until it is committed.

- *Durability or permanency:* Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.



- Suppose the log was last written to disk at t_4.
- By examining the log:
  1. We know that T_0, T_1 and T_2 have committed and their effects should be reflected in the database after recovery.
  2. But we do not know whether the effects of T_0, T_1 and T_2 were reflected at the time of the crash.
  3. We also know that T_3 has started, may have modified some data, but is not committed. Thus T_3 should be undone.

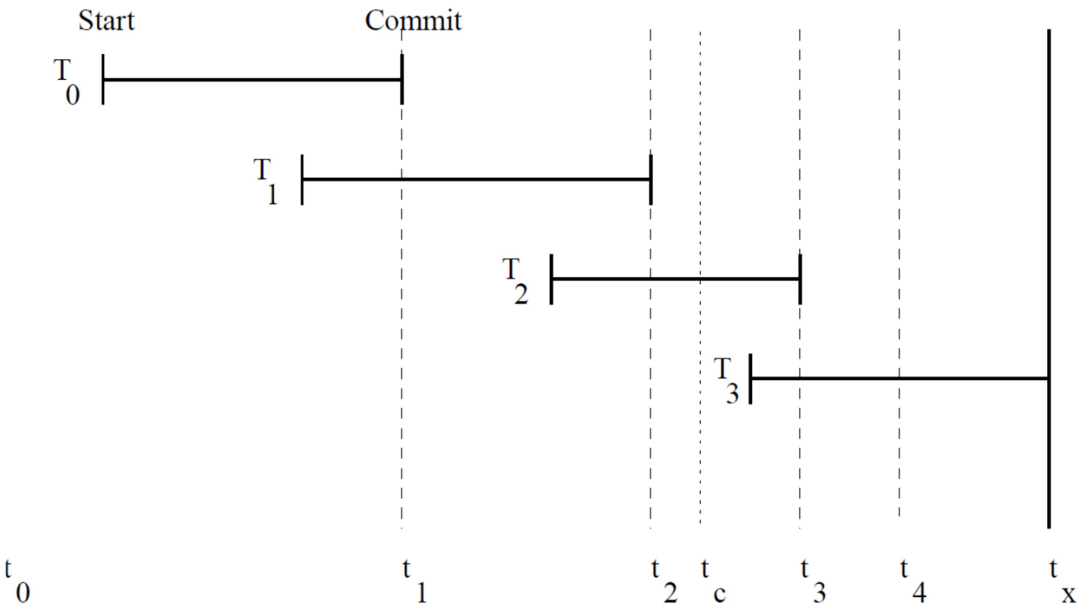
- The database can be recovered by rolling back T_3 using the old data values from the log, and redoing the changes made by T_0 ... T_2 using the new data values (for these committed transactions) from the log.
- Notice that instead of rolling back, the database could have been restored from the backup. This might be necessary in the event of a disk crash for example (for this reason, the log should be stored on an independent disk pack).

Checkpoints

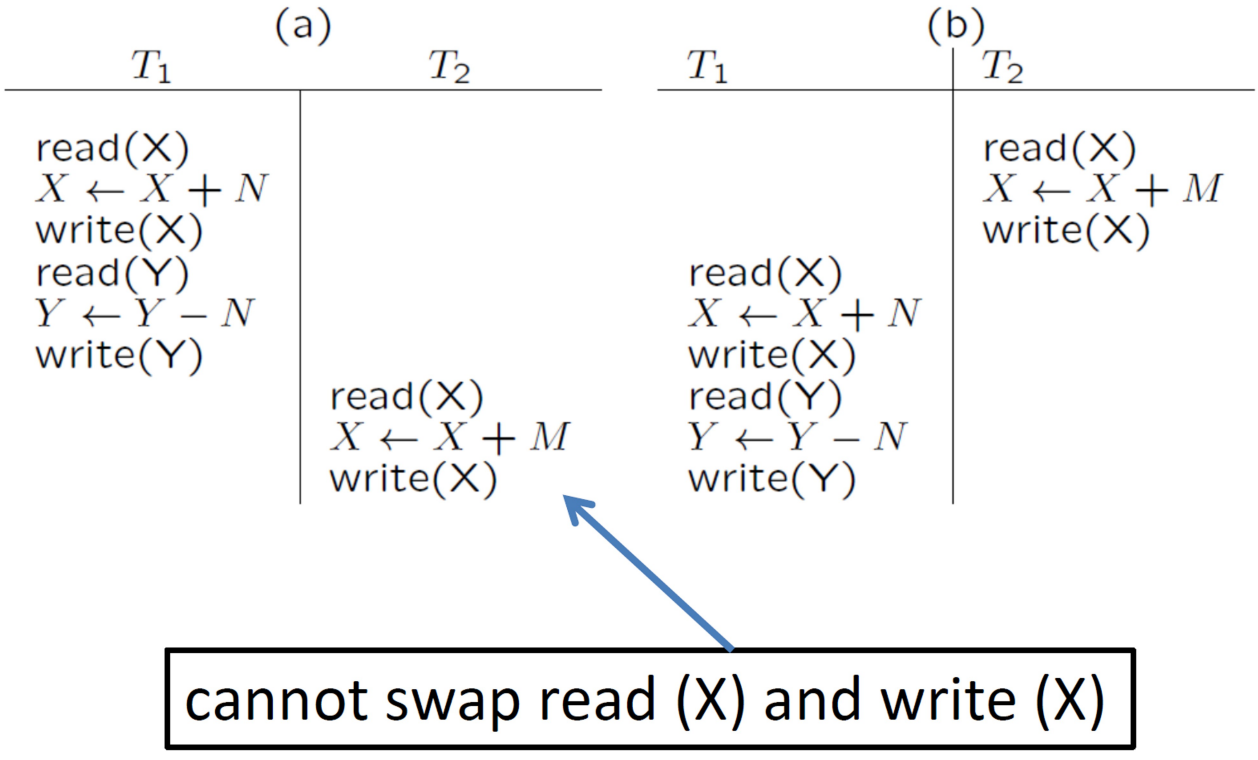
- Notice also that using this system, the longer the time between crashes, the longer recovery may take.

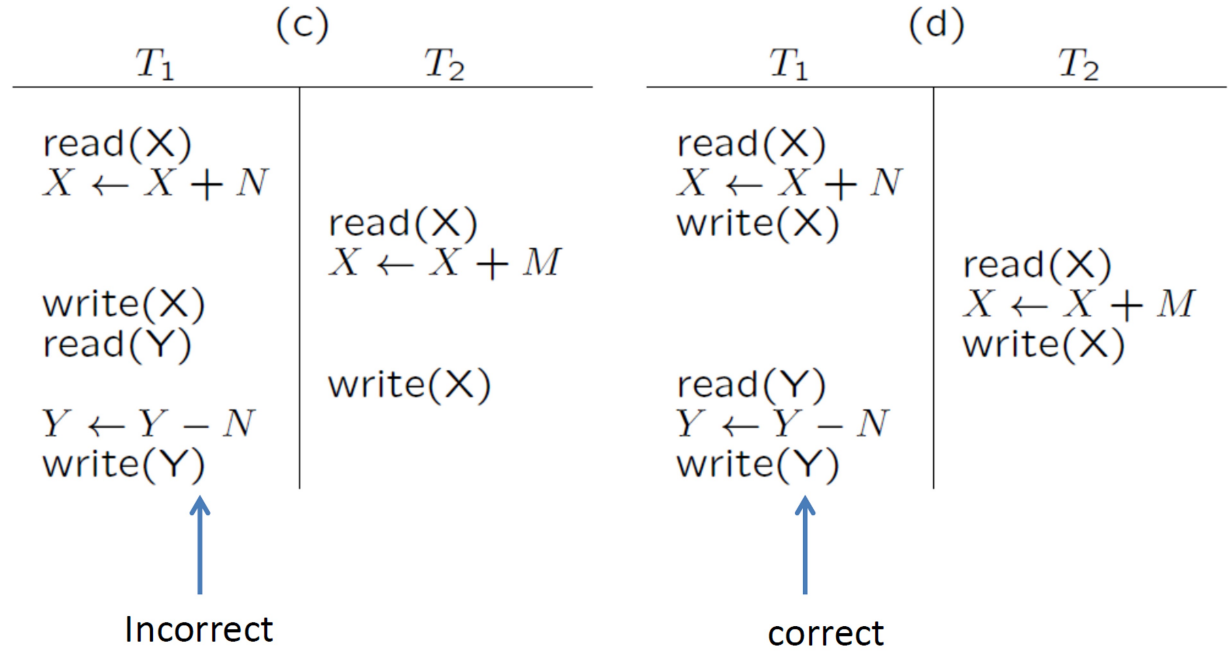
- To avoid this problem, the system may take *checkpoints* at regular intervals.

- To do this:
  - a *start of checkpoint* marker is written to the log, then
  - the database updates in buffers are force-written, then
  - an *end of checkpoint* marker is written to the log.



In our example, suppose a checkpoint is taken at time t_c. Then on recovery we only need redo T_2.



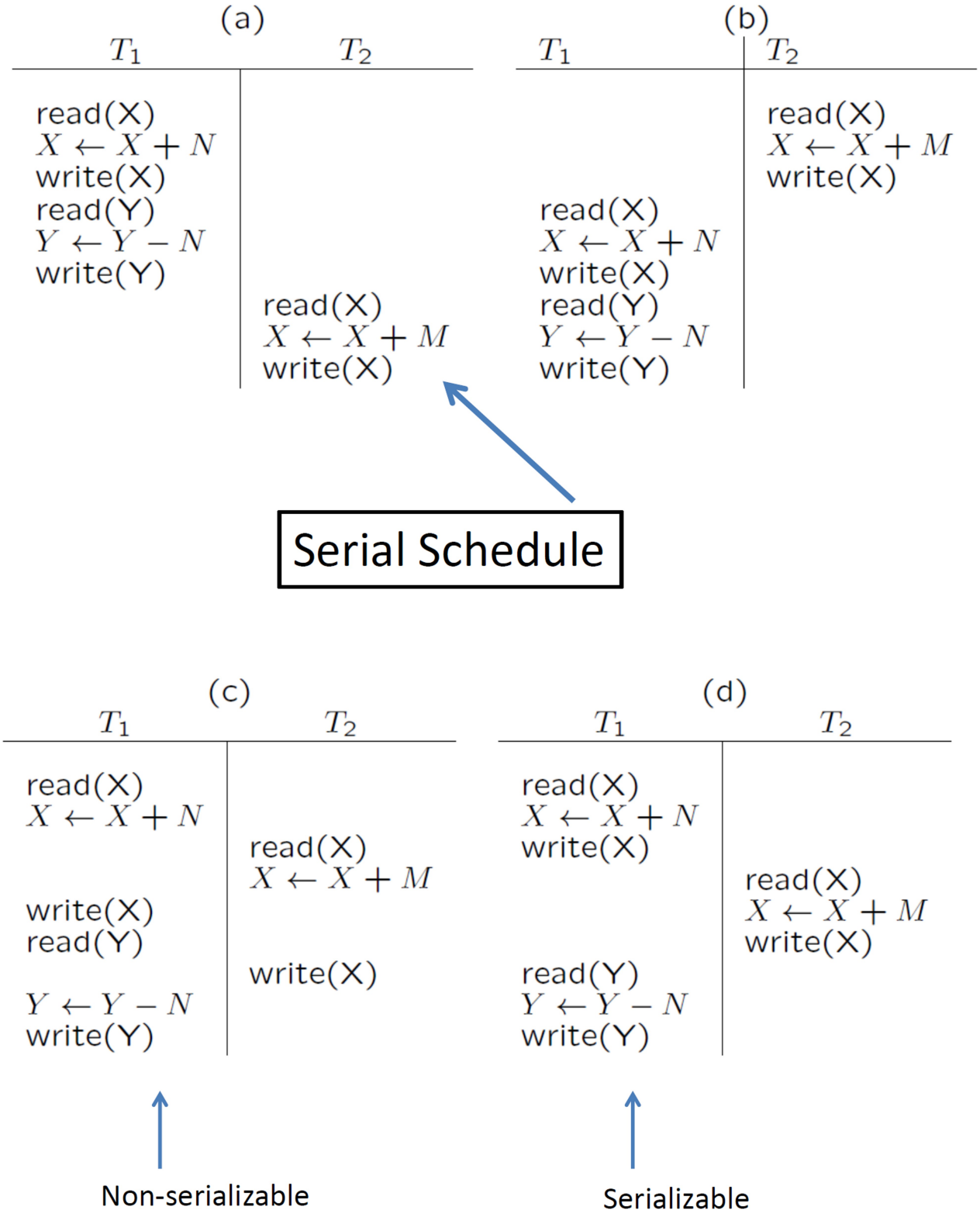


§ As we have seen, if operations are interleaved arbitrarily, incorrect results may occur.
§ However, it is reasonable to assume that schedules (a) and (b) in the figure will give correct results (as long as the transactions are independent).
§ (a) and (b) are called serial schedules, and we will assume that any serial schedule is correct.
§ Notice that schedule (d) always produces the same result as schedules (a) and (b), so it should also give correct results.
§ A schedule is serializable if it always produces the same result as some serial schedule.
§ Notice that schedule (c) is not serializable.

Scheduling Transactions

- Schedule and Complete Schedule?
- Serial schedule: Schedule that does not interleave the actions of different transactions.
- Equivalent schedules: For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.
- Serializable schedule: A schedule over a set S of transactions is equivalent to some serial execution of the set of committed transactions in S.

(Note: If each transaction preserves consistency, every serializable schedule preserves consistency.)


(a)
T_1
T_2
read(X)
$X \leftarrow X + N$
write(X)
read(Y)
$Y \leftarrow Y - N$
write(Y)
read(X)
$X \leftarrow X + M$
write(X)
(b)
T_1
T_2
read(X)
$X \leftarrow X + M$
write(X)
read(X)
$X \leftarrow X + N$
write(X)
read(Y)
$Y \leftarrow Y - N$
write(Y)
Serial Schedule
(c)
T_1
T_2
read(X)
$X \leftarrow X + N$
read(X)
$X \leftarrow X + M$
write(X)
read(Y)
write(X)
$Y \leftarrow Y - N$
write(Y)
Non-serializable
(d)
T_1
T_2
read(X)
$X \leftarrow X + N$
write(X)
read(X)
$X \leftarrow X + M$
write(X)
read(Y)
$Y \leftarrow Y - N$
write(Y)
Serializable

Conflict Serializable Schedules

- Two schedules are *conflict equivalent* if:
  - Involve the same actions of the same transactions
  - Every pair of conflicting actions is ordered the same way
- Schedule S is *conflict serializable* if S is conflict equivalent to some serial schedule

(d)

T_1	T_2
read(X)	
$X \leftarrow X + N$	
write(X)	
	read(X)
	$X \leftarrow X + M$
	write(X)
read(Y)	
$Y \leftarrow Y - N$	
write(Y)	

(a)

T_1	T_2
read(X)	
$X \leftarrow X + N$	
write(X)	
read(Y)	
$Y \leftarrow Y - N$	
write(Y)	
	read(X)
	$X \leftarrow X + M$
	write(X)