RAID

- **RAID: Redundant Arrays of Independent Disks**
  - disk organization techniques that manage a large numbers of disks, providing a view of a single disk of
    - high capacity and high speed by using multiple disks in parallel, and
    - high reliability by storing data redundantly, so that data can be recovered even if a disk fails
- Originally a cost-effective alternative to large, expensive disks
  - I in RAID originally stood for "inexpensive".
  - Today RAIDs are used for their higher reliability and bandwidth.
    - The "I" is interpreted as independent.

Improvement of Reliability via Redundancy

- **Redundancy** – store extra information that can be used to rebuild information lost in a disk failure
- E.g., **Mirroring** (or **shadowing**)
  - Duplicate every disk. Logical disk consists of two physical disks.
  - Every write is carried out on both disks
    - Reads can take place from either disk
  - If one disk in a pair fails, data still available in the other
    - Data loss would occur only if a disk fails, and its mirror disk also fails before the system is repaired
      - Probability of combined event is very small
        - Except for dependent failure modes such as fire or building collapse or electrical power surges
- Mean time to data loss depends on mean time to failure, and mean time to repair

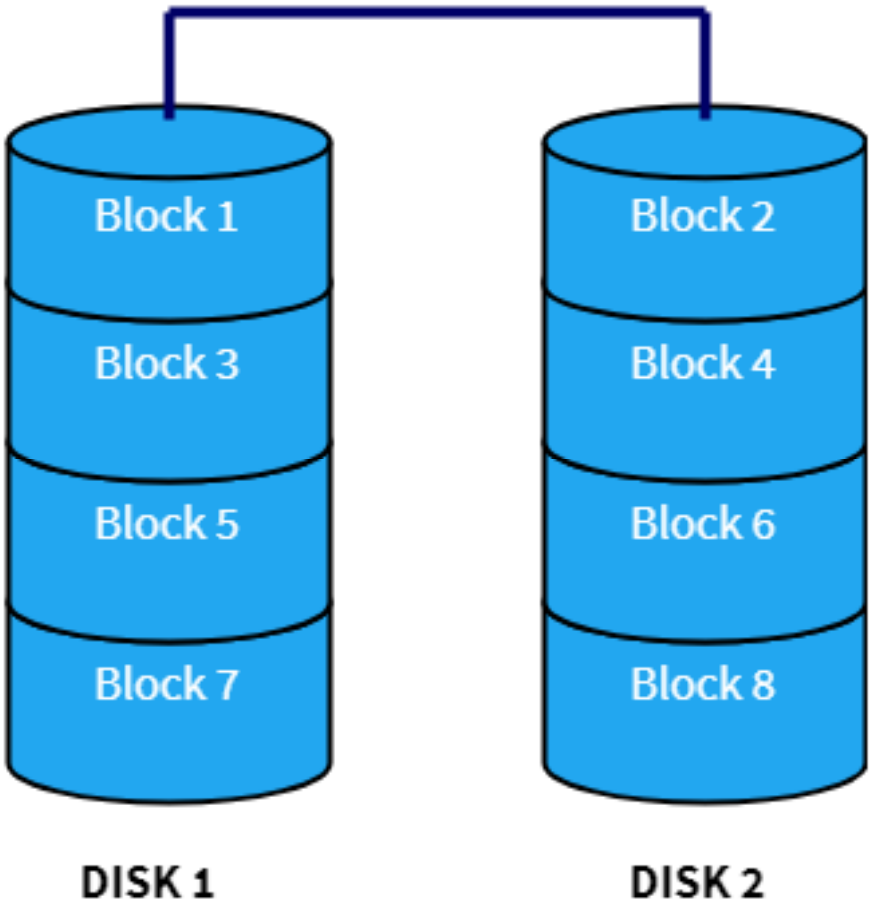
Improvement in Performance via Parallelism

- Two main goals of parallelism in a disk system:
  1. Load balance multiple small accesses to increase throughput
  2. Parallelize large accesses to reduce response time.
- Improve transfer rate by striping data across multiple disks.
- **Bit-level striping** – split the bits of each byte across multiple disks
  - In an array of eight disks, write bit *i* of each byte to disk *i*.
  - Each access can read data at eight times the rate of a single disk.
  - Bit level striping is not used much any more
- **Block-level striping** – with *n* disks, block *i* of a file goes to disk (*i* mod *n*) + 1
  - Requests for different blocks can run in parallel if the blocks reside on different disks
  - A request for a long sequence of blocks can utilize all disks in parallel

RAID 0

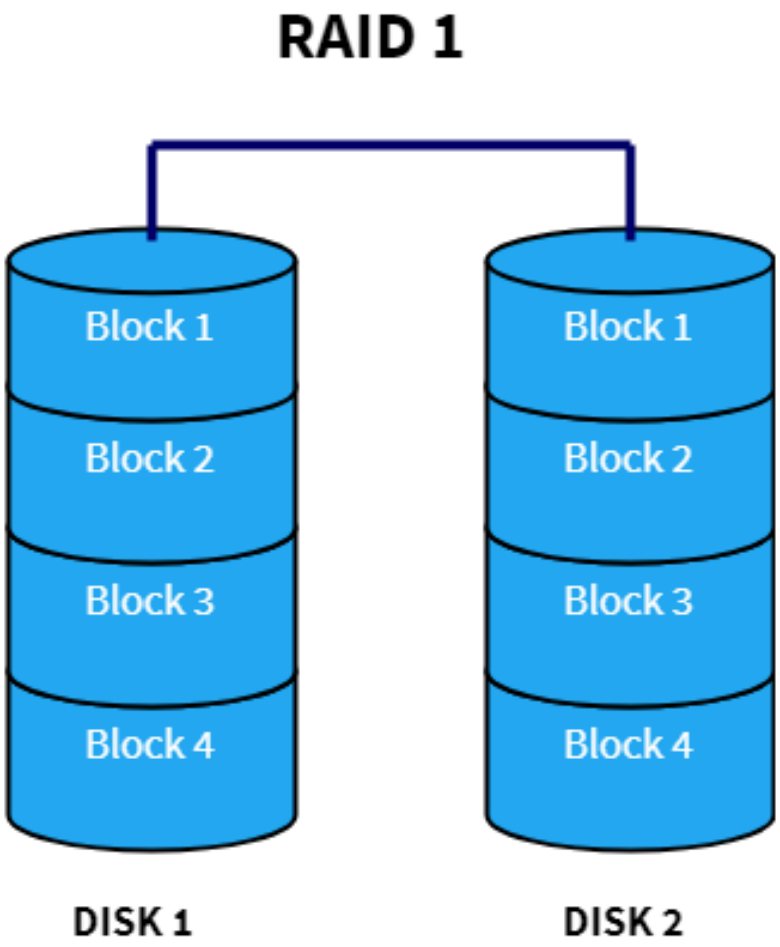
- RAID 0 is based on **data striping**. A stream of data is divided into multiple segments or blocks and each of those blocks is stored on different disks. So, when the system wants to read that data, it can do so simultaneously from all the disks and join them together to reconstruct the entire data stream.
- The benefit of this is that the **speed increases** drastically for read and write operations. It is great for situations where **performance** is a priority over other aspects.
- Also, the total capacity of the entire volume is the sum of the capacities of the individual disks.
- There is almost **no redundancy**. If one of the disks fails, the entire data becomes corrupt and worthless since it cannot be recreated anymore.
- Advantages:
  - Performance boost for read and write operations
  - Space is not wasted as the entire volume of the individual disks are used up to store unique data
- Disadvantages
  - There is no redundancy/duplication of data. If one of the disks fails, the entire data is lost.

RAID 0


Block 1
Block 3
Block 5
Block 7
Block 2
Block 4
Block 6
Block 8
DISK 1
DISK 2

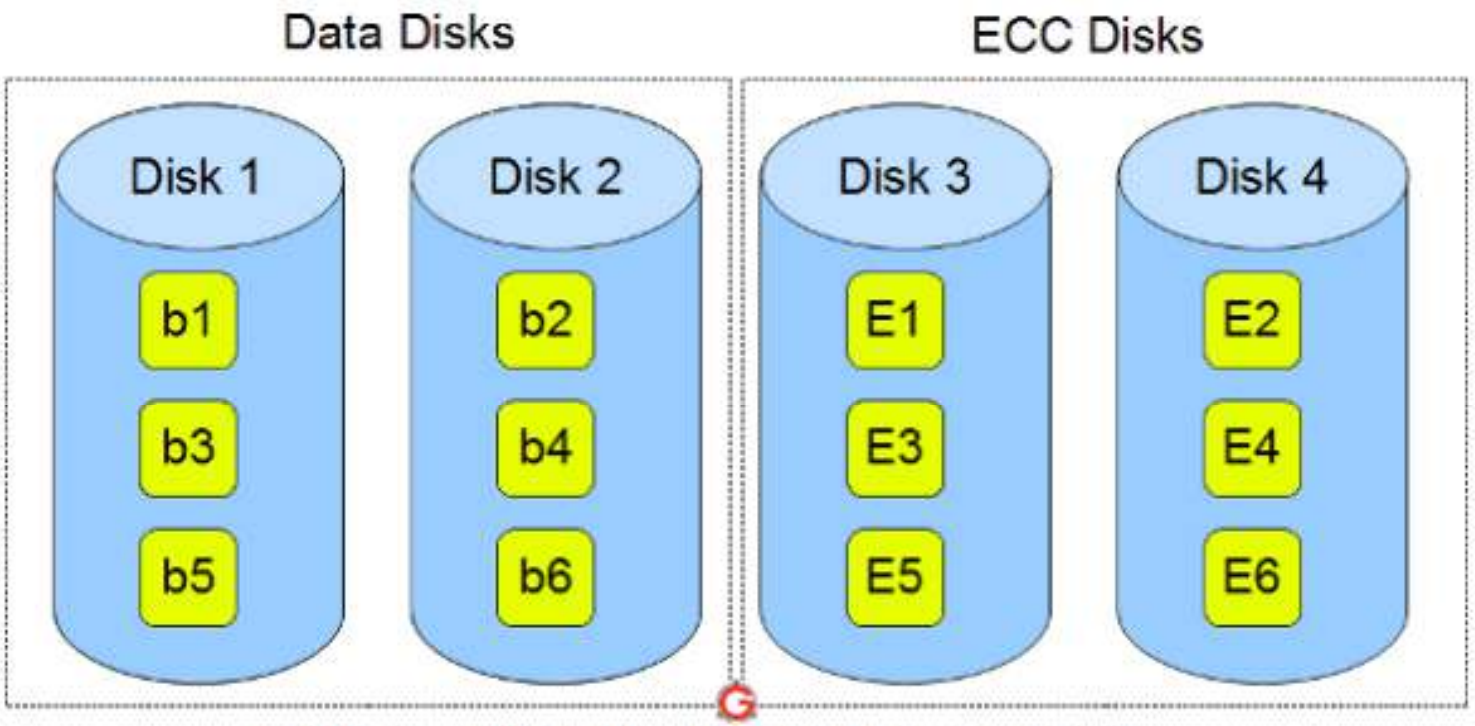
RAID 1

- RAID 1 uses the concept of **data mirroring**. Data is mirrored or cloned to an identical set of disks so that if one of the disks fails, the other one can be used.
- It also **improves read performance** since different blocks of data can be accessed from all the disks simultaneously. This can be explained in the diagram. A multi-threaded process can access Block 1 from Disk 1 and Block 2 from Disk 2 at once thereby increasing the read speed just like RAID 0.
- But unlike RAID 0, **write performance is reduced** since all the drives must be updated whenever new data is written.
- Another disadvantage is that **space is wasted** to duplicate the data thereby increasing the cost to storage ratio.
- Advantages
  - Data can be recovered in case of disk failure
  - Increased performance for read operation
- Disadvantages
  - Slow write performance
  - Space is wasted by duplicating data which increases the cost per unit memory


RAID 1
Block 1
Block 2
Block 3
Block 4
Block 1
Block 2
Block 3
Block 4
DISK 1
DISK 2

RAID 2

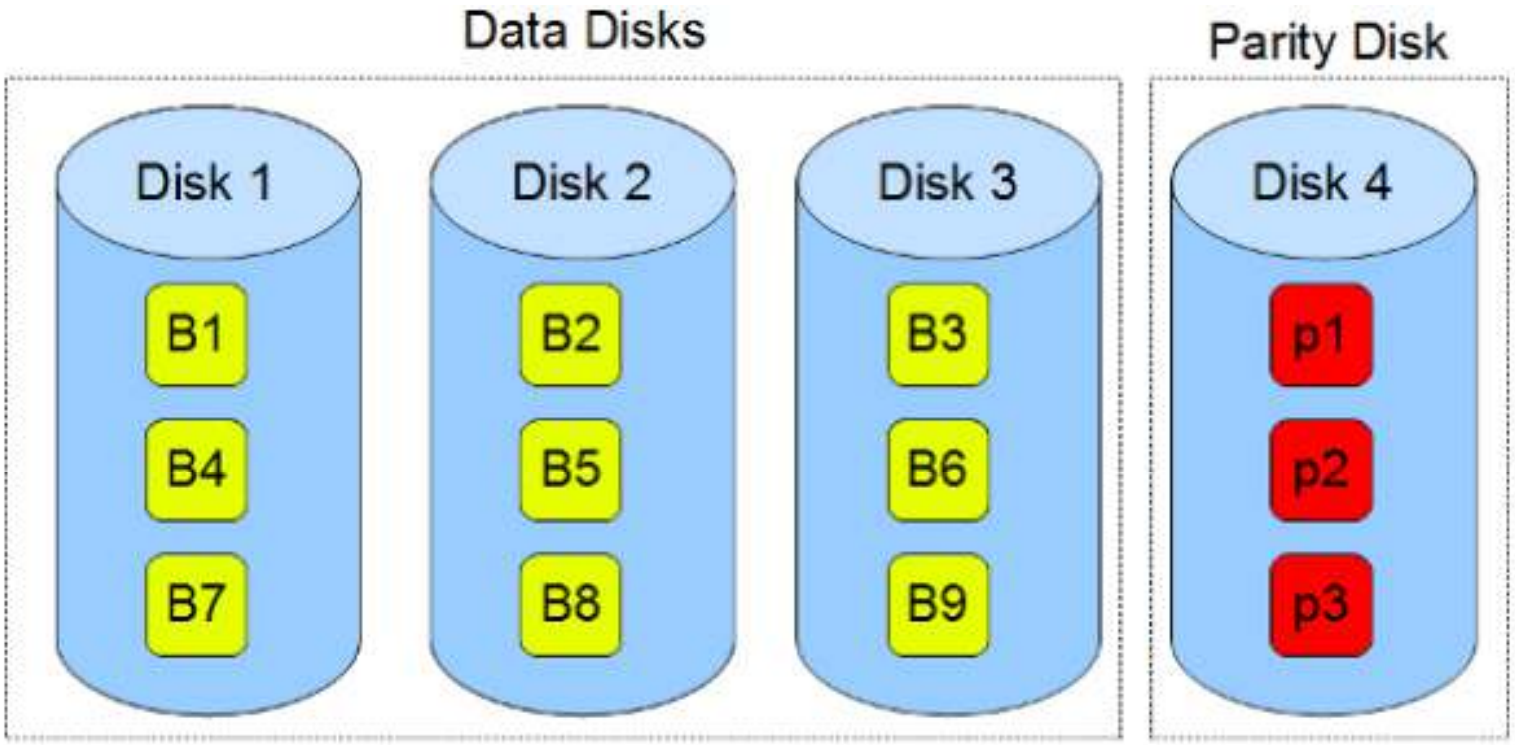
- This uses **bit level striping** i.e instead of striping the blocks across the disks, it stripes the bits across the disks.
- In the next diagram b1, b2, b3 are bits. E1, E2, E3 are error correction codes.
- You need two groups of disks. One group of disks are used to write the data, another group is used to write the error correction codes.
- This uses Hamming error correction code (ECC), and stores this information in the redundancy disks.
- When data is written to the disks, it calculates the ECC code for the data on the fly, and stripes the data bits to the data-disks, and writes the ECC code to the redundancy disks.
- When data is read from the disks, it also reads the corresponding ECC code from the redundancy disks, and checks whether the data is consistent. If required, it makes appropriate corrections on the fly.
- This uses lot of disks and can be configured in different disk configuration. Some valid configurations are 1) 10 disks for data and 4 disks for ECC 2) 4 disks for data and 3 disks for ECC
- **This is not used anymore**. This is expensive and implementing it in a RAID controller is complex, and ECC is redundant now-a-days, as the hard disk themselves can do this.



RAID 2 – Bits Striped. (and stores ECC)

RAID 3

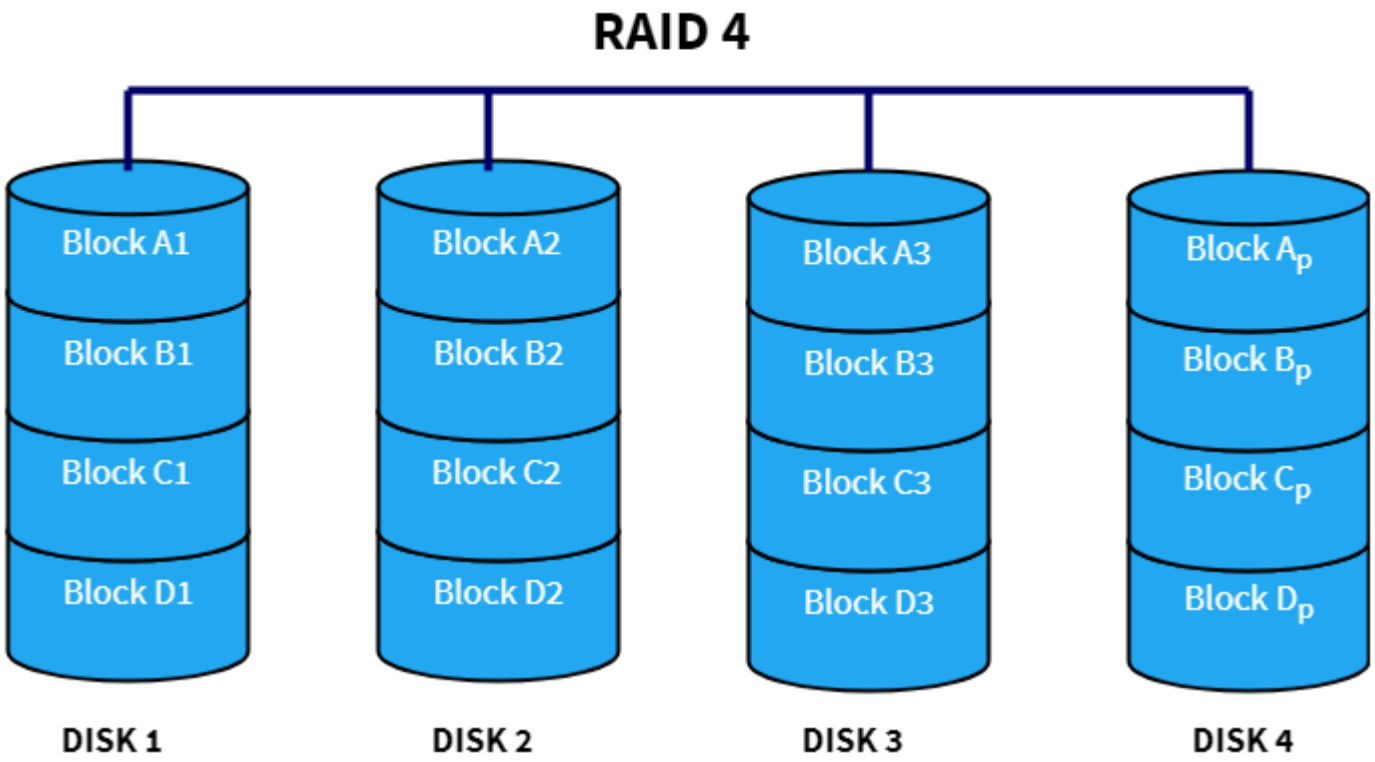
- This uses bit level striping. i.e Instead of striping the blocks across the disks, it stripes the bytes across the disks.
- In the diagram B1, B2, B3 are bytes. p1, p2, p3 are parities.
- Uses multiple data disks, and a dedicated disk to store parity.
- The disks have to spin in sync to get to the data.
- Sequential read and write will have good performance.
- Random read and write will have worst performance.
- This is not commonly used.



RAID 3 – Bytes Striped. (and Dedicated Parity Disk)

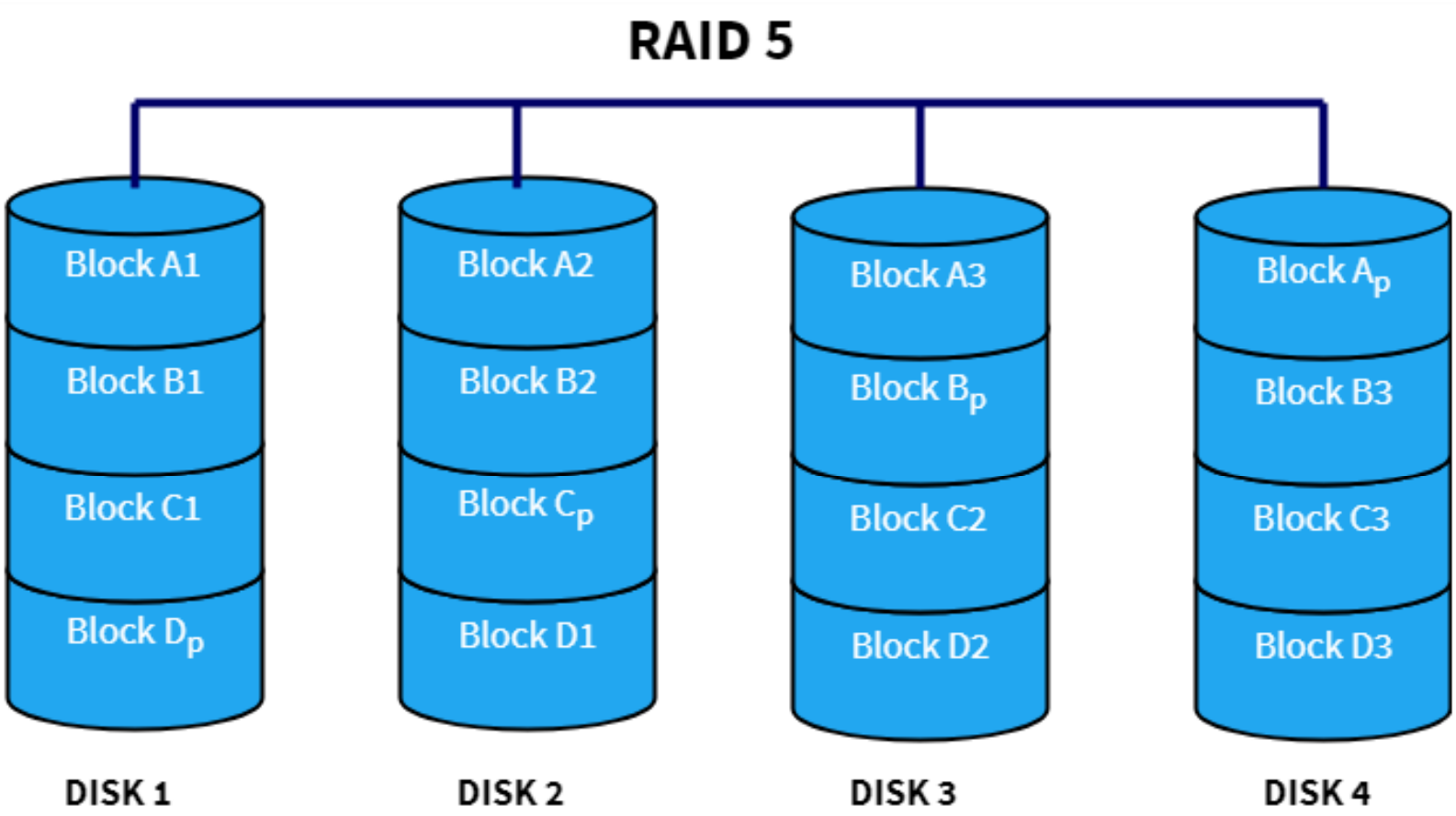
RAID 4

- RAID 4 stripes the data across multiple disks just like RAID 0.
- In addition to that, it also stores parity information of all the disks in a separate dedicated disk to achieve redundancy.
- In the diagram, Disk 4 serves as the parity disk having parity blocks A_p, B_p, C_p and D_p. So, if one of the disks fails, the data can be reconstructed using the parity information of that disk.
- **Space is more efficiently used here when compared to RAID 1 since parity information uses way less space than mirroring the disk.**
- The write performance becomes slow because all the parity information is written on a single disk which is a bottleneck. This problem is solved in RAID 5 as we will see next.
- Advantages
  - Efficient data redundancy in terms of cost per unit memory
  - Performance boost for read operations due to data stripping
- Disadvantages
  - Write operation is slow
  - If the dedicated parity disk fails, data redundancy is lost


RAID 4
Block A1
Block B1
Block C1
Block D1
Block A2
Block B2
Block C2
Block D2
Block A3
Block B3
Block C3
Block D3
Block A_p
Block B_p
Block C_p
Block D_p
DISK 1
DISK 2
DISK 3
DISK 4

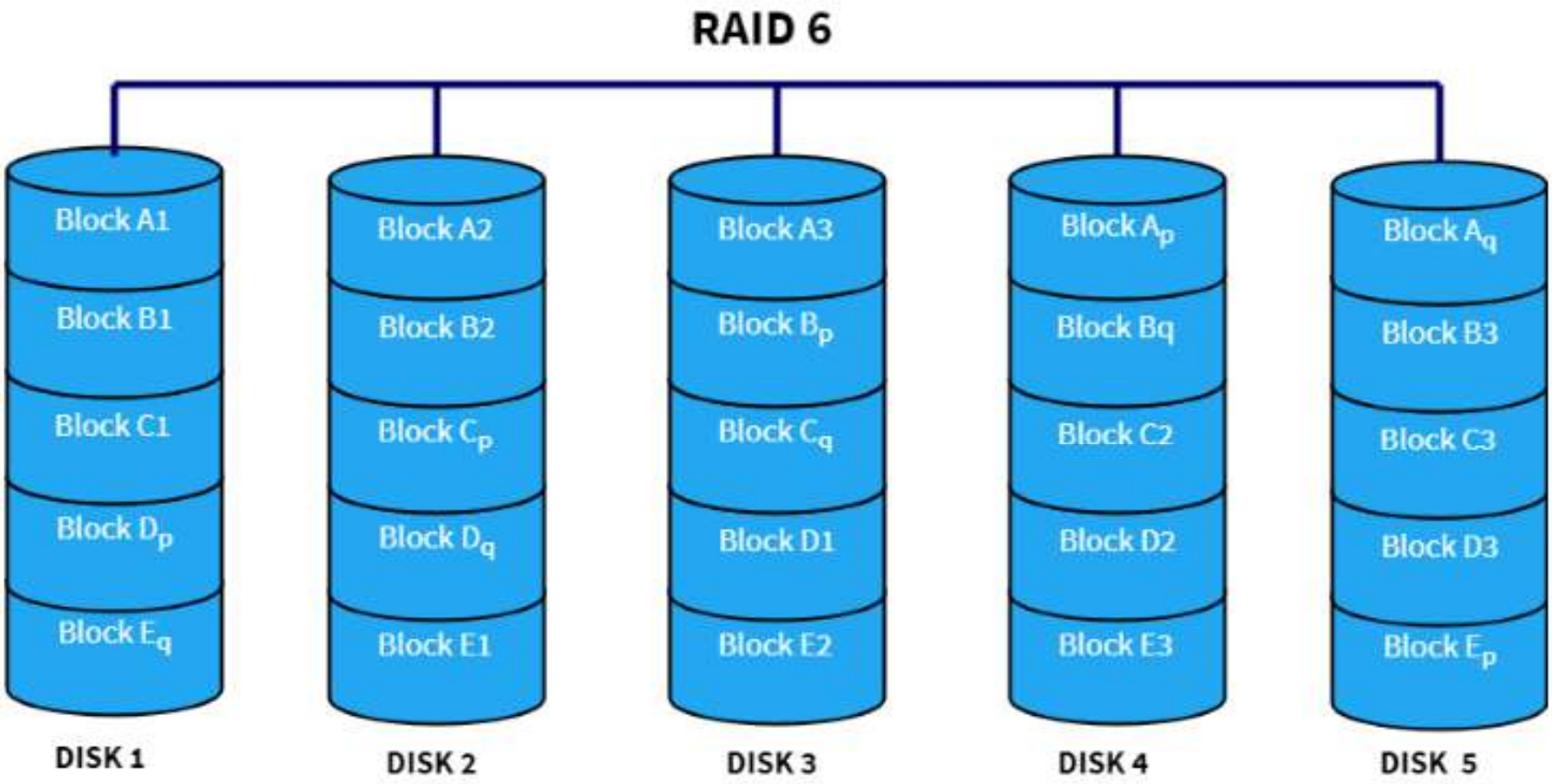
RAID 5

- RAID 5 is very similar to RAID 4, but here the parity information is distributed over all the disks instead of storing them in a dedicated disk.
- This has two benefits — First, there is no more a bottleneck as the parity stress evens out by using all the disks to store parity information.
- And second, there is no possibility of losing data redundancy since one disk does not store all the parity information.
- Advantages
  - All the advantages of RAID 4 plus increased write speed and better data redundancy. Additionally, write performance is increased since all RAID members participate in the serving of write requests.
- Disadvantages
  - Can only handle up to a single disk failure


RAID 5
Block A1
Block B1
Block C1
Block D_p
Block A2
Block B2
Block C_p
Block D1
Block A3
Block B_p
Block C2
Block D2
Block A_p
Block B3
Block C3
Block D3
DISK 1
DISK 2
DISK 3
DISK 4

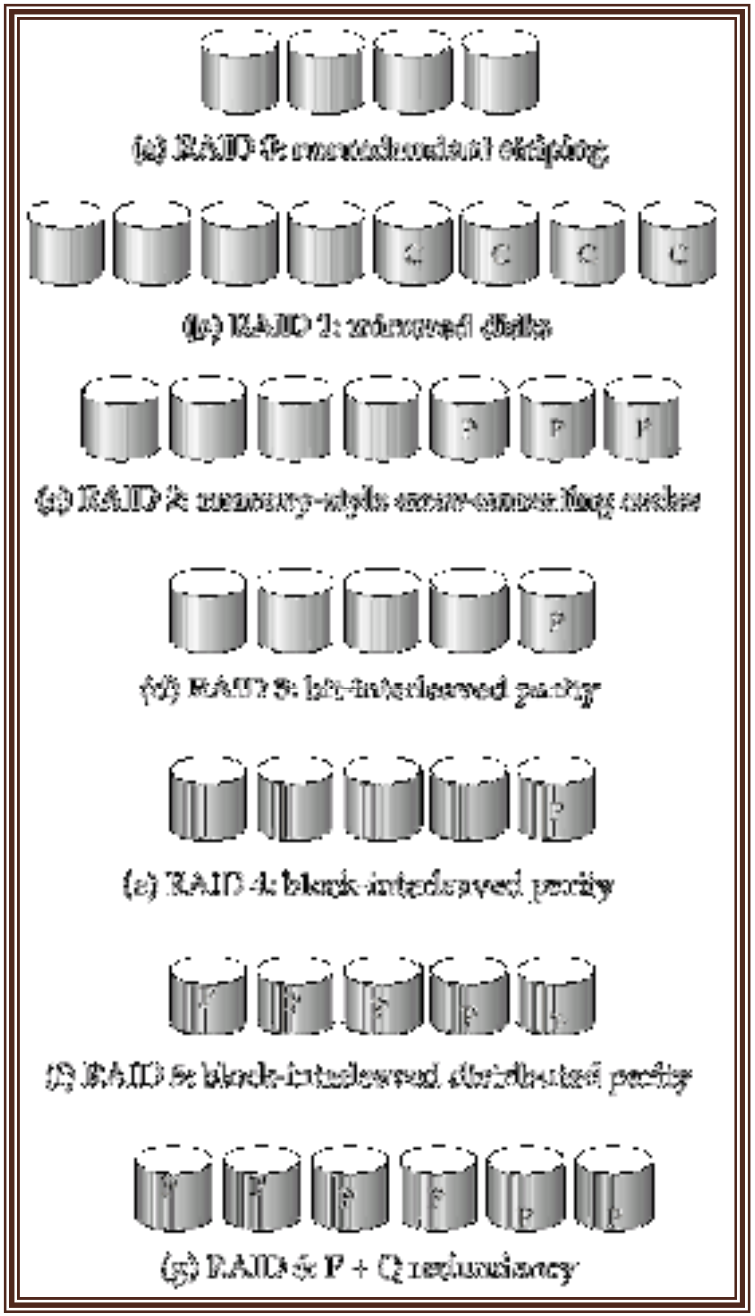
RAID 6

- RAID 6 uses double parity blocks to achieve better data redundancy than RAID 5. This increases the fault tolerance for upto two drive failures in the array.
- Each disk has two parity blocks which are stored on different disks across the array. RAID 6 is a very practical infrastructure for maintaining high availability systems.
- Advantages
  - Better data redundancy. Can handle up to 2 failed drives or Can handle two disk failure
- Disadvantages
  - Large parity overhead


RAID 6
Block A1
Block B1
Block C1
Block D_p
Block E_q
Block A2
Block B2
Block C_p
Block D_q
Block E1
Block A3
Block B_p
Block C_q
Block D1
Block E2
Block A_p
Block Bq
Block C2
Block D2
Block E3
Block A_q
Block B3
Block C3
Block D3
Block E_p
DISK 1
DISK 2
DISK 3
DISK 4
DISK 5

Choice of RAID Level

- Factors in choosing RAID level
  - Monetary cost
  - Performance: Number of I/O operations per second, and bandwidth during normal operation
  - Performance during failure
  - Performance during rebuild of failed disk
    - Including time taken to rebuild failed disk
- RAID 0 is used only when data safety is not important
  - E.g. data can be recovered quickly from other sources
- Level 2 and 4 never used since they are subsumed by 3 and 5
- Level 3 is not used anymore since bit-striping forces single block reads to access all disks, wasting disk arm movement, which block striping (level 5) avoids
- Level 6 is rarely used since levels 1 and 5 offer adequate safety for almost all applications
- So competition is between 1 and 5 only


(a) RAID 0: nonredundant striping
C
(b) RAID 1: mirrored disks
P
(c) RAID 2: memory-style error-correcting codes
(d) RAID 3: bit-interleaved parity
(e) RAID 4: block-interleaved parity
(f) RAID 5: block-interleaved distributed parity
Q
(g) RAID 6: P + Q redundancy

File Organization

- The database is stored as a collection of *files*. Each file is a sequence of *records*. A record is a sequence of fields.
- One approach:
  - assume record size is fixed
  - each file has records of one particular type only
  - different files are used for different relations

  This case is easiest to implement; will consider variable length records later.

Fixed-Length Records

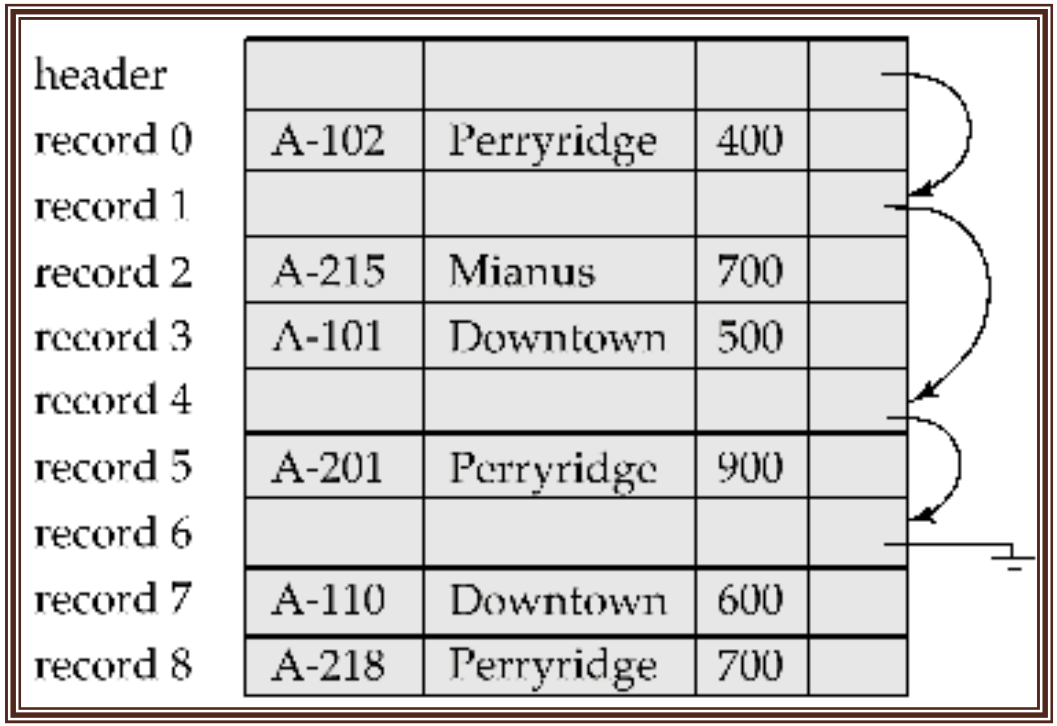
- Simple approach:
  - Record access is simple.
- Deletion of record *I*: alternatives:
  - move records $i + 1, \ldots, n$ to $i, \ldots, n - 1$

  OR
  - do not move records, but link all free records on a *free list*

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 2	A-215	Mianus	700
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

Free Lists

- Store the address of the first deleted record in the file header.
- Use this first record to store the address of the second deleted record, and so on
- Can think of these stored addresses as pointers since they "point" to the location of a record.



Variable-Length Records

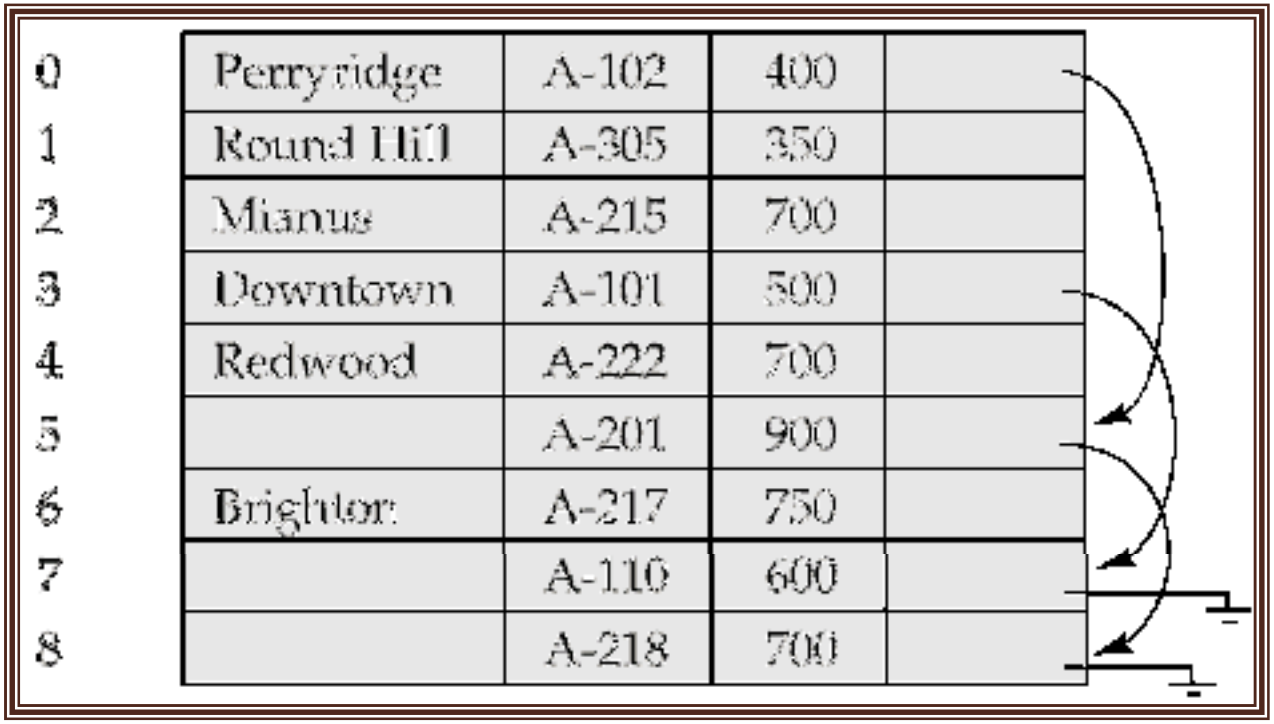
- Variable-length records arise in database systems in several ways:
  - Storage of multiple record types in a file.
  - Record types that allow variable lengths for one or more fields.
  - Record types that allow repeating fields (used in some older data models).
- Byte string representation
  - Attach an *end-of-record* (⊥) control character to the end of each record
  - Difficulty with deletion
  - Difficulty with growth

Variable-Length Records (Cont.)

- Fixed-length representation:
  - reserved space
  - pointers
- Reserved space – can use fixed-length records of a known maximum length; unused space in shorter records filled with a null or end-of-record symbol.

0	Perryridge	A-102	400	A-201	900	A-218	700
1	Round Hill	A-305	350	⊥	⊥	⊥	⊥
2	Mianus	A-215	700	⊥	⊥	⊥	⊥
3	Downtown	A-101	500	A-110	600	⊥	⊥
4	Redwood	A-222	700	⊥	⊥	⊥	⊥
5	Brighton	A-217	750	⊥	⊥	⊥	⊥

Pointer Method



0	Perryridge	A-102	400	
1	Round Hill	A-305	350	
2	Mianus	A-215	700	
3	Downtown	A-101	500	
4	Redwood	A-222	700	
5		A-201	900	
6	Brighton	A-217	750	
7		A-110	600	
8		A-218	700	

- Pointer method
  - A variable-length record is represented by a list of fixed-length records, chained together via pointers.
  - Can be used even if the maximum record length is not known

Pointer Method (Cont.)

- Disadvantage to pointer structure; space is wasted in all records except the first in chain.
- Solution is to allow two kinds of block in file:
  - Anchor block – contains the first records of chain
  - Overflow block – contains records other than those that are the first records of chairs.

Organization of Records in Files

- **Heap** – a record can be placed anywhere in the file where there is space
- **Sequential** – store records in sequential order, based on the value of the search key of each record
- **Hashing** – a hash function computed on some attribute of each record; the result specifies in which block of the file the record should be placed
- Records of each relation may be stored in a separate file. In a **clustering file organization** records of several different relations can be stored in the same file
  - Motivation: **store related records on the same block to minimize I/O**

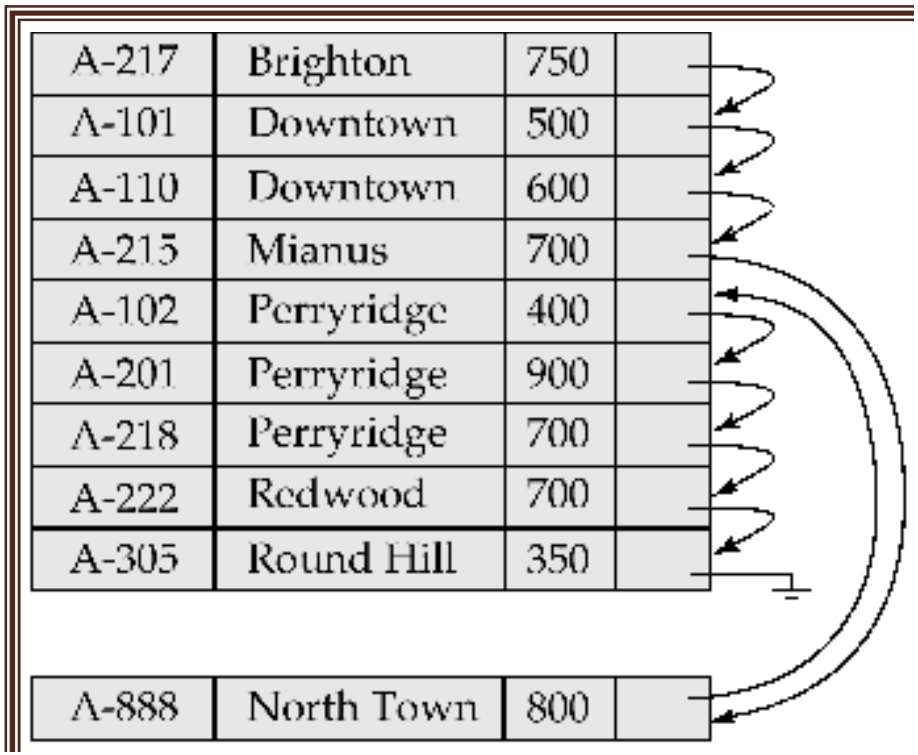
Sequential File Organization

- Suitable for applications that require sequential processing of the entire file
- The records in the file are ordered by a search-key

A-217	Brighton	750	
A-101	Downtown	500	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	

Sequential File Organization (Cont.)

- Deletion – use pointer chains
- Insertion –locate the position where the record is to be inserted
  - if there is free space insert there
  - if no free space, insert the record in an overflow block
  - In either case, pointer chain must be updated
- Need to reorganize the file from time to time to restore sequential order



Clustering File Organization

- Simple file structure stores each relation in a separate file
- Can instead store several relations in one file using a **clustering** file organization
- E.g., clustering organization of *customer* and *depositor:*

Hayes	Main	Brooklyn
Hayes	A-102	
Hayes	A-220	
Hayes	A-503	
Turner	Putnam	Stamford
Turner	A-305	

- good for queries involving depositor ⋈ customer, and for queries involving one single customer and his accounts
- bad for queries involving only customer

Failure Classification

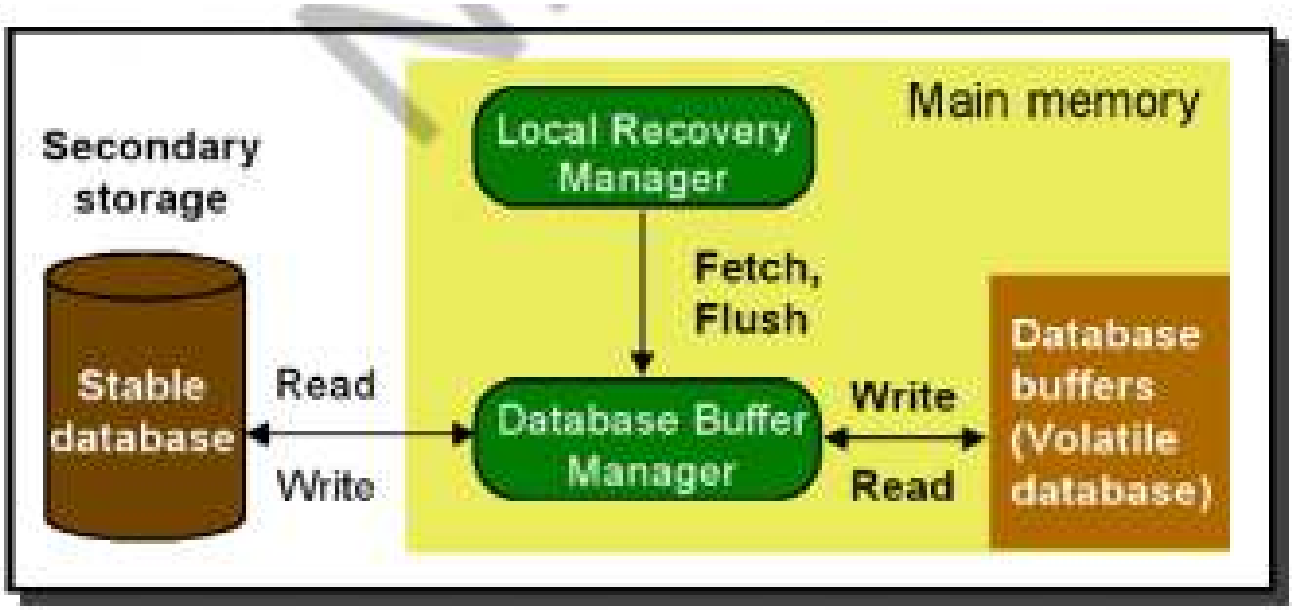
- **Transaction failure** :
  - **Logical errors**: transaction cannot complete due to some internal error condition
  - **System errors**: the database system must terminate an active transaction due to an error condition (e.g., deadlock)
- **System crash**: a power failure or other hardware or software 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
- **Disk failure**: a head crash or similar disk failure destroys all or part of disk storage

Recovery Algorithms

- Recovery algorithms are techniques to ensure database consistency and transaction atomicity and durability despite failures
- Recovery algorithms have two parts
  1. Actions taken **during normal transaction** processing to ensure enough information exists to recover from failures
  2. Actions taken **after a failure to recover** the database contents to a state that ensures atomicity, consistency and durability

Storage Structure

- **Volatile storage**:
  - does not survive system crashes
  - examples: main memory, cache memory
- **Nonvolatile storage**:
  - survives system crashes
  - examples: disk, tape, flash memory, non-volatile (battery backed up) RAM
- **Stable storage**:
  - A form of storage that survives all failures
  - approximated by maintaining multiple copies on distinct nonvolatile media



Stable-Storage Implementation

- Maintain multiple copies of each block on separate disks
  - copies can be at remote sites to protect against disasters such as fire or flooding.
- Failure during data transfer can still result in inconsistent copies: Block transfer can result in
  - Successful completion
  - Partial failure: destination block has incorrect information
  - Total failure: destination block was never updated
- Protecting storage media from failure during data transfer (one solution):
  - Execute output operation as follows (assuming two copies of each block):
    1. Write the information onto the first physical block.
    2. When the first write successfully completes, write the same information onto the second physical block.
    3. The output is completed only after the second write successfully completes.

Stable-Storage Implementation (Cont.)

- Protecting storage media from failure during data transfer (cont.):
- Copies of a block may differ due to failure during output operation. To recover from failure:
  1. First find inconsistent blocks:
     1. *Expensive solution*: Compare the two copies of every disk block.
     2. *Better solution*:
        - Record in-progress disk writes on non-volatile storage (Non-volatile RAM or special area of disk).
        - Use this information during recovery to find blocks that may be inconsistent, and only compare copies of these.
        - Used in hardware RAID systems
  2. If either copy of an inconsistent block is detected to have an error (bad checksum), overwrite it by the other copy. If both have no error, but are different, overwrite the second block by the first block.

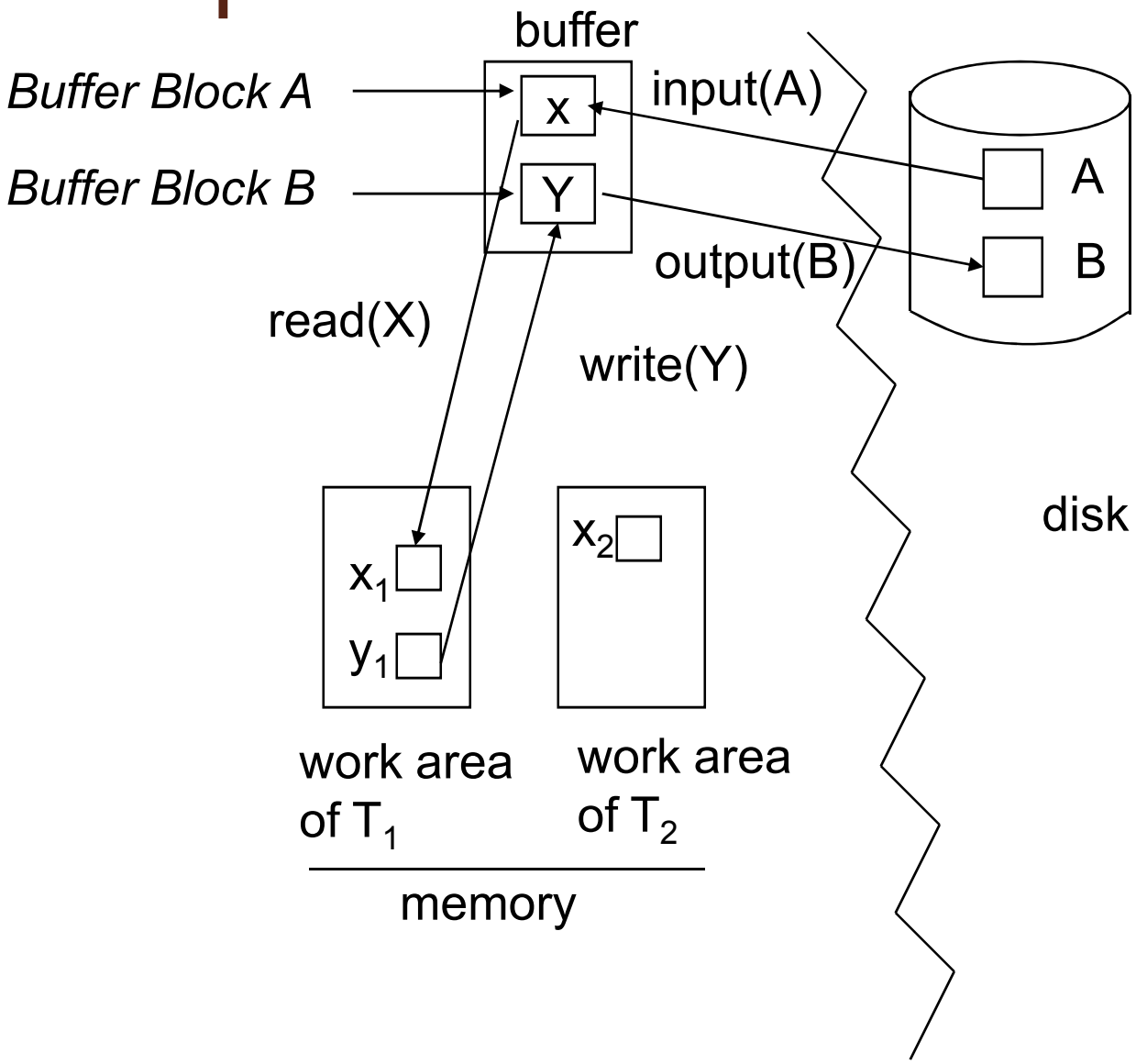
Data Access

- **Physical blocks** are those blocks residing on the disk.
- **Buffer blocks** are the blocks residing temporarily in main memory.
- Block movements between disk and main memory are initiated through the following two operations:
  - **input**(*B*) transfers the physical block *B* to main memory.
  - **output**(*B*) transfers the buffer block *B* to the disk, and replaces the appropriate physical block there.
- Each transaction T_i has its private work-area in which local copies of all data items accessed and updated by it are kept.
  - T_i's local copy of a data item *X* is called x_i.
- We assume, for simplicity, that each data item fits in, and is stored inside, a single block.

Data Access (Cont.)

- Transaction transfers data items between system buffer blocks and its private work-area using the following operations :
  - **read**(X) assigns the value of data item X to the local variable x_i.
  - **write**(X) assigns the value of local variable x_i to data item {X} in the buffer block.
  - both these commands may necessitate the issue of an **input**(B_X) instruction before the assignment, if the block B_X in which X resides is not already in memory.
- Transactions
  - Perform **read**(X) while accessing X for the first time;
  - All subsequent accesses are to the local copy.
  - After last access, transaction executes **write**(X).
- **output**(B_X) need not immediately follow **write**(X). System can perform the **output** operation when it deems fit.

Example of Data Access



Recovery and Atomicity

- Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state.
- Consider transaction T_i that transfers \$50 from account *A* to account *B*; goal is either to perform all database modifications made by T_i or none at all.
- Several output operations may be required for T_i (to output *A* and *B*). A failure may occur after one of these modifications have been made but before all of them are made.

Recovery and Atomicity (Cont.)

- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.
- We study two approaches:
  - **log-based recovery**, and
  - **shadow-paging**
- We assume (initially) that transactions run serially, that is, one after the other.

Log-Based Recovery

- A **log** is kept on stable storage.
  - The log is a sequence of **log records**, and maintains a record of update activities on the database.
- When transaction T_i starts, it registers itself by writing a
  $<T_i$ **start**>log record
- *Before* T_i executes **write**(X), a log record $<T_i, X, V_1, V_2>$ is written,

  where V_1 is the value of X before the write, and V_2 is the value to be written to X.
  - Log record notes that T_i has performed a write on data item X_j X_j had value V_1 before the write, and will have value V_2 after the write.
- When T_i finishes it last statement, the log record $<T_i$ **commit**> is written.
- We assume for now that log records are written directly to stable storage (that is, they are not buffered)
- Two approaches using logs
  - Deferred database modification
  - Immediate database modification

Deferred Database Modification

- The **deferred database modification** scheme records all modifications to the log, but defers all the **write**s to after partial commit.
- Assume that transactions execute serially
- Transaction starts by writing <T_i ***start***> record to log.
- A **write**(X) operation results in a log record <T_i, X, V> being written, where V is the new value for X
  - Note: old value is not needed for this scheme
- The write is not performed on X at this time, but is deferred.
- When T_i partially commits, <T_i **commit**> is written to the log
- Finally, the log records are read and used to actually execute the previously deferred writes.

Deferred Database Modification (Cont.)

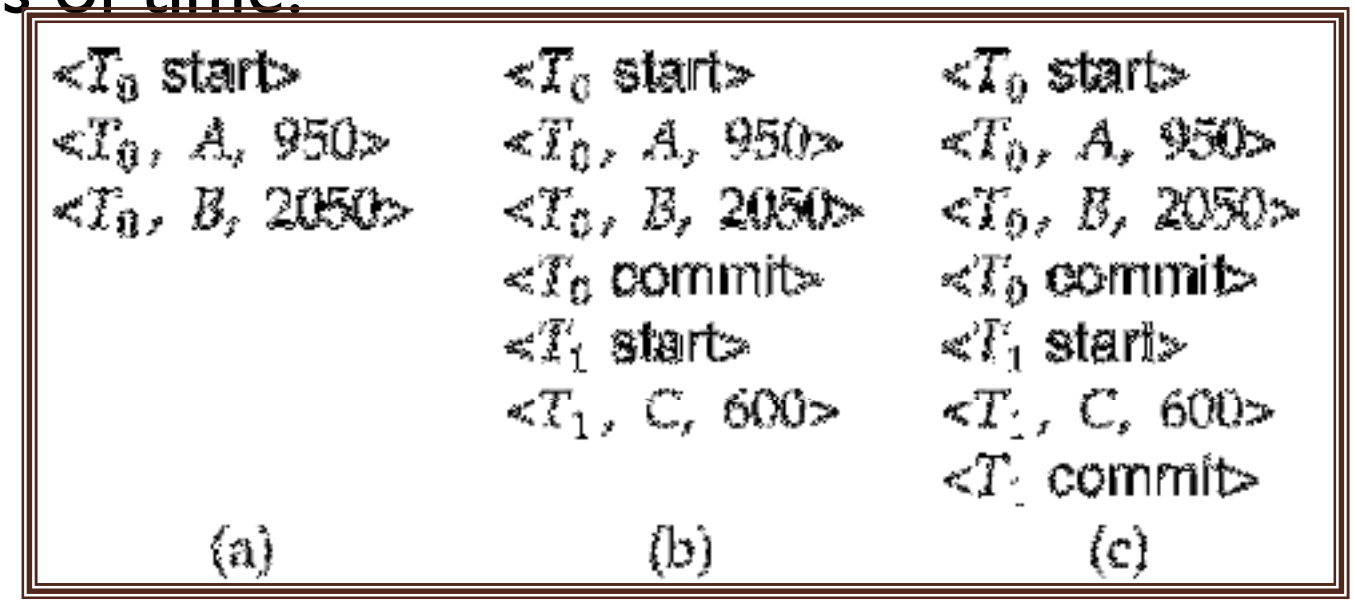
- During recovery after a crash, a transaction needs to be **redone** if and only if both <T_i **start**> and<T_i **commit**> are there in the log.
- Redoing a transaction T_i (**redo** T_i) sets the value of all data items updated by the transaction to the new values.
- Crashes can occur while
  - the transaction is executing the original updates, or
  - while recovery action is being taken
- example transactions T_0 and T_1 (T_0 executes before T_1):

T_0: **read** (A)
A: - A - 50
Write (A)
read (B)
B:- B + 50
write (B)

T_1 : **read** (C)
C=C- 100
write (C)

Deferred Database Modification

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



- 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 **commit**> is present

(c) **redo**(T_0) must be performed followed by redo(T_1) since <T_0 **commit**> and <T_i commit> are present

Immediate Database Modification

- The **immediate database modification** scheme allows database updates of an uncommitted transaction to be made as the writes are issued
  - since undoing may be needed, update logs must have both old value and new value
- Update log record must be written *before* database item is written
  - We assume that the **log record is output directly to stable storage**
  - 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
- Output of updated blocks can take place at any time before or after transaction commit

Immediate Database Modification Example

Log	Write	Output
<T_0 **start**>		
<T_0, A, 1000, 950>		
<T_0, B, 2000, 2050>		
	$A = 950$	
	$B = 2050$	
<T_0 **commit**>		
<T_1 **start**>		
<T_1, C, 700, 600>		
	$C = 600$	
<T_1 **commit**>		

Immediate Database Modification

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

Immediate DB Modification Recovery Example

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

(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

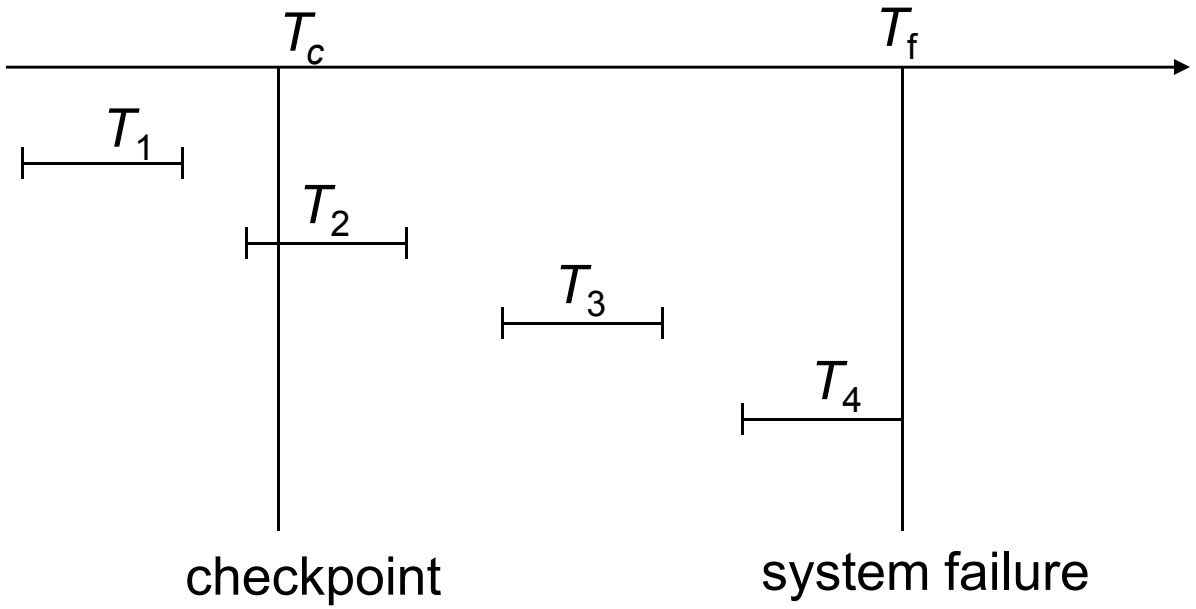
Checkpoints

- Problems in recovery procedure as discussed earlier :
  1. searching the entire log is time-consuming
  2. we might unnecessarily redo transactions which have already output their updates to the database.
- Streamline recovery procedure by periodically performing **checkpointing**
  1. Output all log records currently residing in main memory onto stable storage.
  2. Output all modified buffer blocks to the disk.
  3. Write a log record < **checkpoint**> onto stable storage.

Checkpoints (Cont.)

- During recovery we need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i.
  1. Scan backwards from end of log to find the most recent <**checkpoint**> record
  2. Continue scanning backwards till a record <T_i **start**> is found.
  3. 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.
  4. For all transactions (starting from T_i or later) with no <T_i **commit**>, execute **undo(***T_i***)**. (Done only in case of immediate modification.)
  5. Scanning forward in the log, for all transactions starting from T_i or later with a <T_i **commit**>, execute **redo(***T_i***)**.

Example of Checkpoints

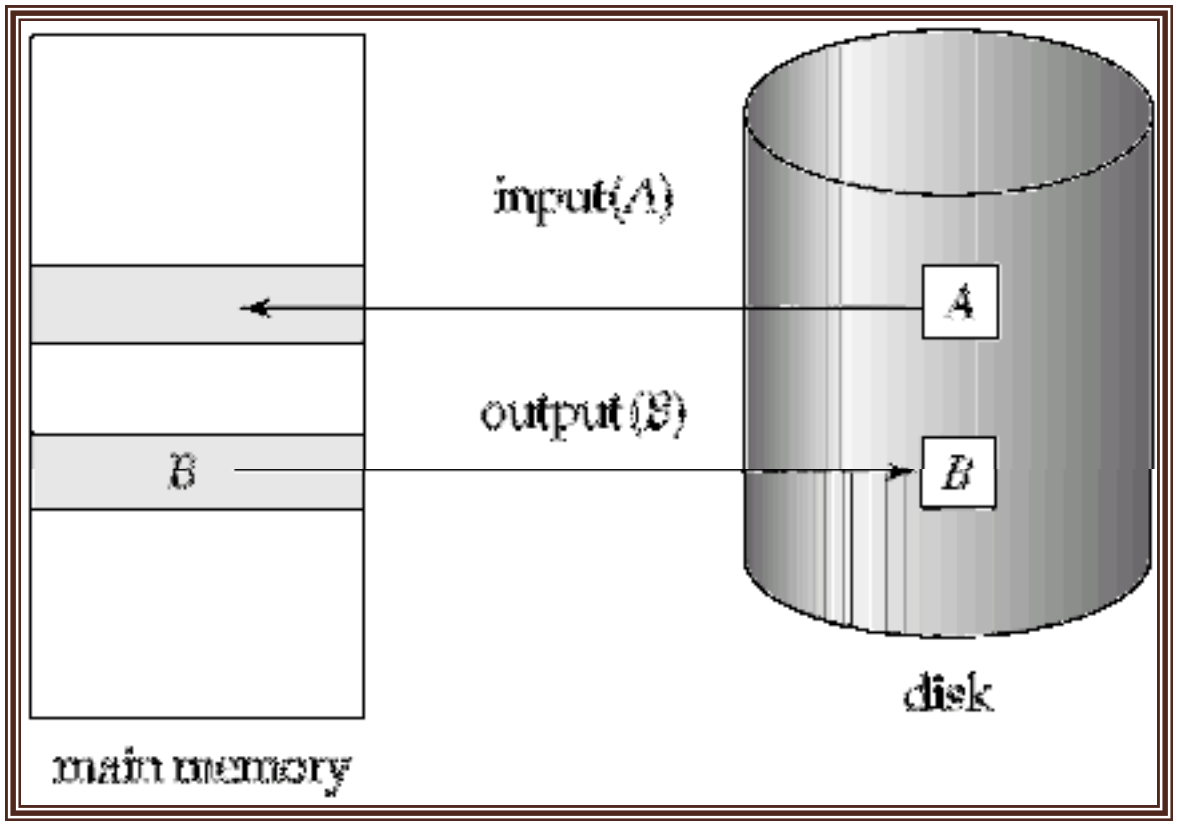


- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone.
- T_4 undone

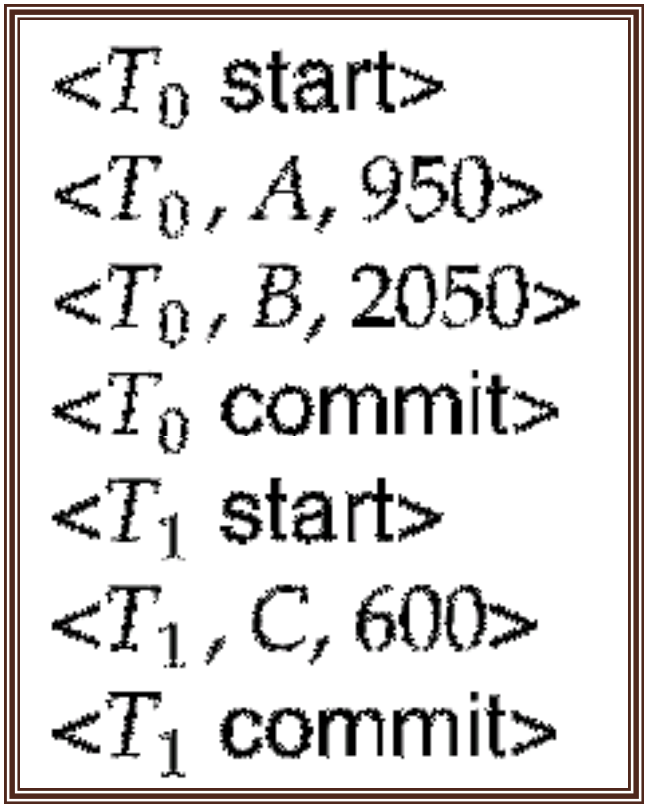
Shadow Paging

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

Block Storage Operations



Portion of the Database Log Corresponding to T_0 and T_1



State of the Log and Database Corresponding to T_0 and T_1

Log	Database
$<T_0$ start$>$	
$<T_0, A, 950>$	
$<T_0, B, 2050>$	
$<T_0$ commit$>$	
	$A = 950$
	$B = 2050$
$<T_1$ start$>$	
$<T_1, C, 600>$	
$<T_1$ commit$>$	
	$C = 600$

Portion of the System Log Corresponding to T_0 and T_1

<T_0 start>
<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>
<T_0 commit>
<T_1 start>
<T_1, C, 700, 600>
<T_1 commit>

State of System Log and Database Corresponding to T_0 and T_1

Log	Database
$<T_0$ start$>$	
$<T_0, A, 1000, 950>$	
$<T_0, B, 2000, 2050>$	
	$A = 950$
	$B = 2050$
$<T_0$ commit$>$	
$<T_1$ start$>$	
$<T_1, C, 700, 600>$	
	$C = 600$
$<T_1$ commit$>$	