Storing Data

What are we storing?

- Relations comprising tables
- Auxiliary structures: indices (or indexes)
- Tables can be partitioned in several ways
- Think of each table partition as a file
- File is typically a collection of data blocks (pages)
- Pages store the rows (or its partitions)

Persistent Storage

- Data in a DBMS has to be persistent
- Persistent storage options
  - SSD: flash memory based, faster and more expensive
  - HDD: magnetic storage, slower and cheaper
  - Tapes: disks are the new tapes, still great for archiving
  - Cloud?
  - Backup?

Volatile Storage

- RAM is required for processing data
- Would unlimited memory make SSD/HDD obsolete?
- What if RAM was cheaper than SSD?

Memory Hierarchy

- Tape/Cloud could be called "tertiary storage"
- Data persists in SDD/HDD/combo aka "secondary storage"
- Brought to RAM (primary storage) for processing
- Brought to on-chip cache automatically, but algorithms can affect it's utilization
- Rest is automatic

Registers

Cache

RAM

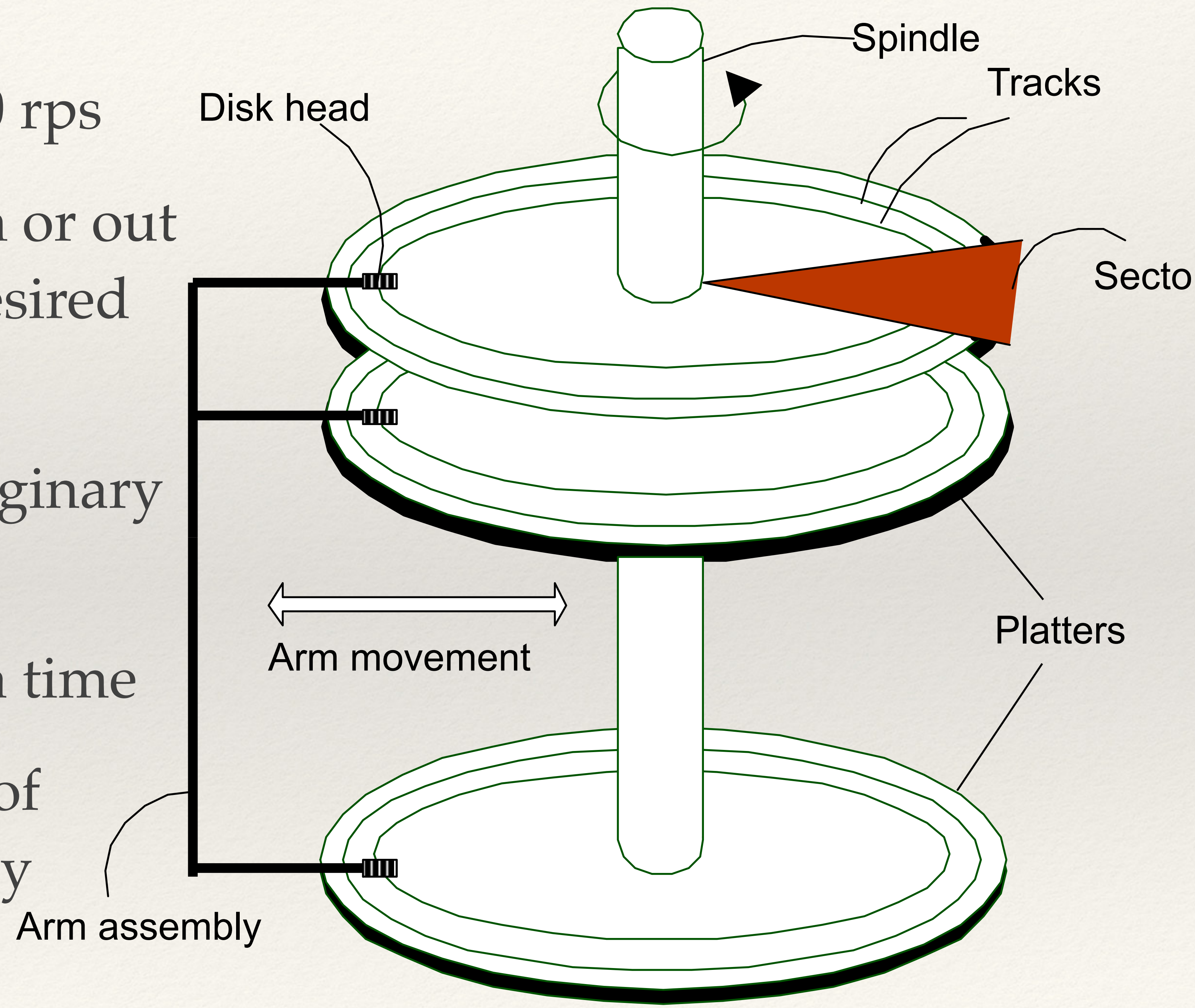
SSD

HDD

Tapes (Rare)

Disk Components

- The platters spin, say 90 rps
- Arm assembly moves in or out to position a head on desired track.
- The tracks make an imaginary cylinder
- Only one head used at a time
- Block size is a multiple of sector size (fixed, usually 512 bytes)



Disk Page Access

- Time to access (read/write) a disk block:
  - seek time (moving arms to position disk head on track)
  - rotational delay (waiting for block to rotate under head)
  - transfer time (actually moving data to/from disk surface)
- Seek time and rotational delay dominate.
  - Seek time varies from about 1 to 20msec
  - Rotational delay varies from 0 to 10msec
  - Transfer rate is about 1msec per 4KB page
- Key to lower I/O cost: reduce seek/rotation delays!

To minimize seek/rotation delay

- Next block should be on:
    - same track, (no seek, no rotation), then
    - same cylinder, (no seek, possibly rotation), then
    - adjacent cylinder, (minimal seek, possibly rotation)
- Blocks of a file are arranged "sequentially" as above
- Prefetching blocks in a sequential scan is a big win

Disk Space Management

- Lowest layer of DBMS software manages space on disk.
- Higher levels call upon this layer to:
  - allocate/de-allocate a pageread/write a page
- Request for a sequence of pages must be satisfied by allocating the pages sequentially on disk.
- Higher levels don't need to know how this is done, or how free space is managed.

What about SSD?

- Flash based SSD's are now common
- No seek or rotational latency, better I/O performance
- Different read/write behavior
  - writes to same blocks progressively slower
  - mitigated by wear leveling
- DBMS use it for storing (caching) "hot" data

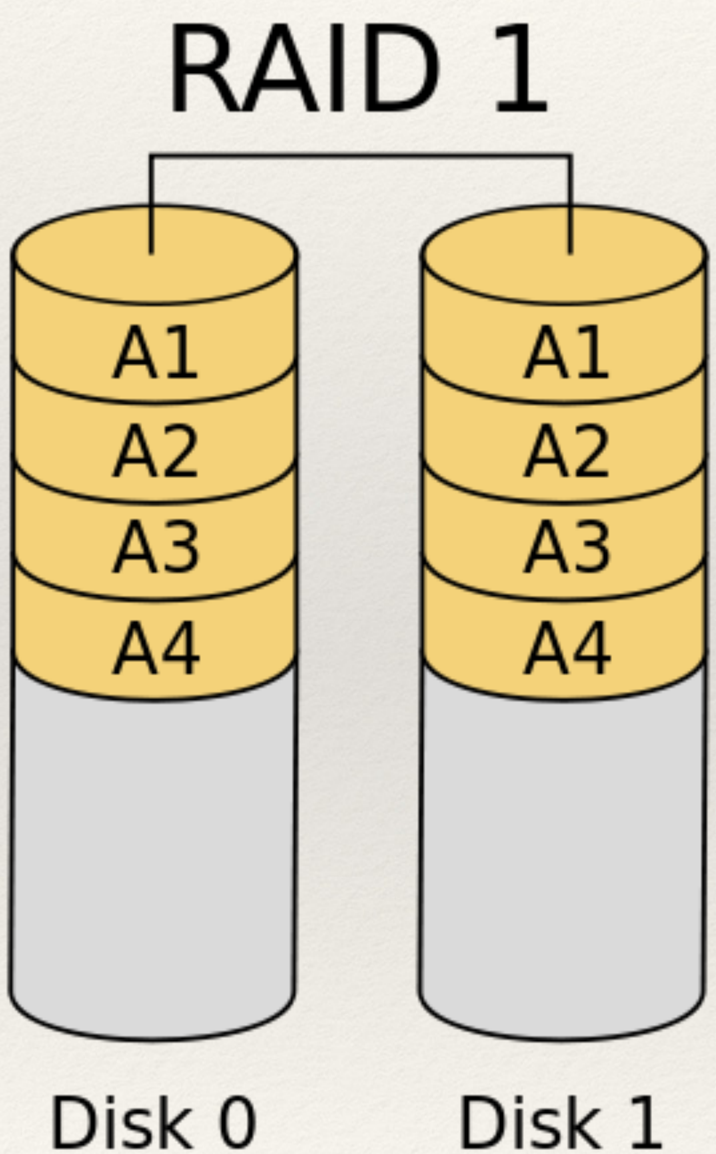
SSD/HDD comparison

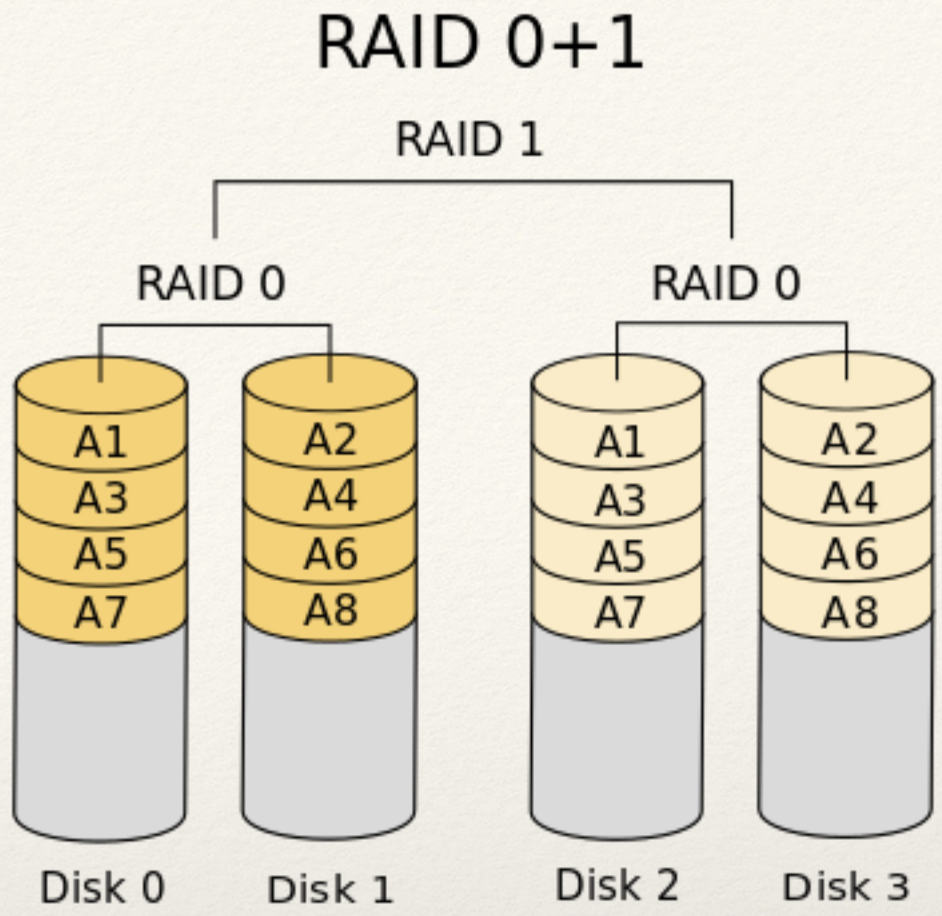
Attribute	SSD	HDD
Random Access Time	Typ. under 0.1 ms	2.9ms - 12ms
Read Latency	very low	much higher, depends on seek/rotational delay
Data Transfer Rate	100MB/s-600MB/s	140MB/s at high end
Read Performance	independent of location	varies depending on access pattern
Write Performance	slightly slower and gradually worsens	about same and doesn't get worse
Reliability	despite write issues comparably reliable	better offline storage shelf-life
Cost	getting cheaper	cheaper still

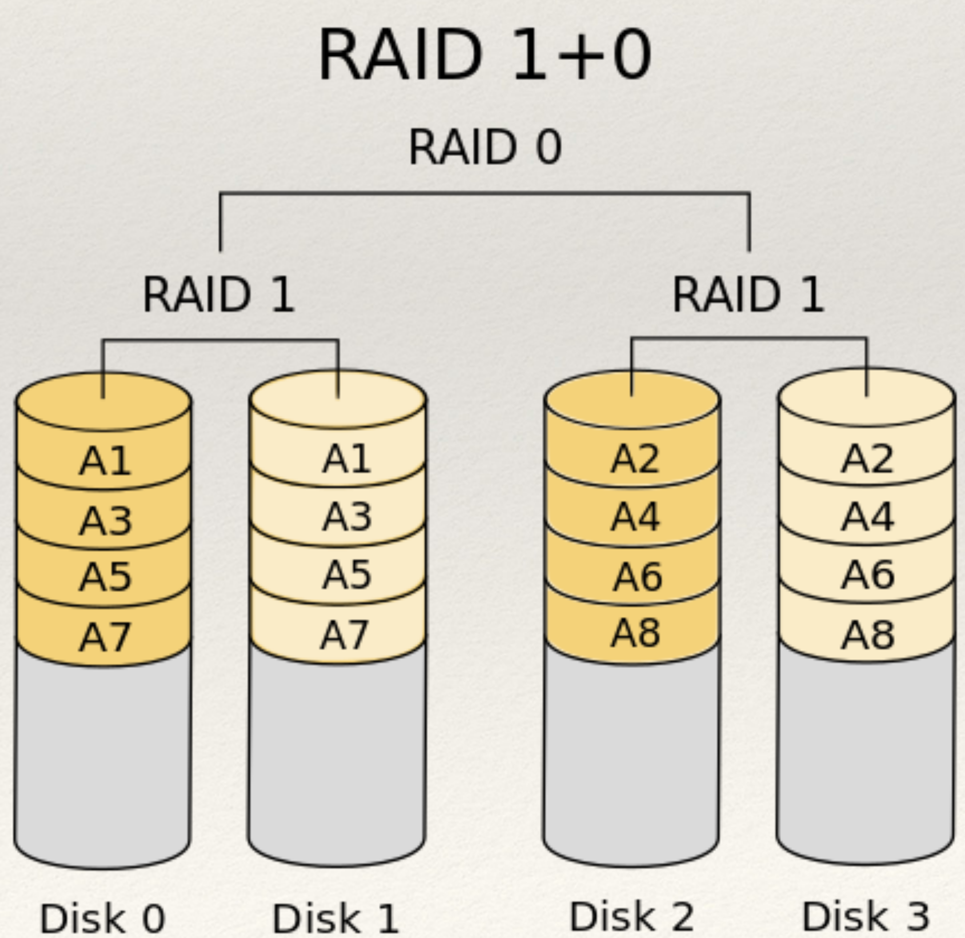
RAID

- Use multiple disks together to improve read performance and reliability
- Failure of 1 (or more) disk doesn't lose data
- Read can use multiple disks simultaneously
- RAID 0: data striping, contiguous blocks of data on different disks
- RAID 1: mirror data on a different disk, half usable capacity, half write performance, but can suffer loss of more than 1 disk
  - RAID 0+1 "striped mirror" , RAID 1+0 "mirrored stripe"
- RAID 5: keep parity information on a separate disk so that data from loss of 1 disk can be reconstructed from others
  - more usable capacity, but less reliable than RAID 1
  - RAID 6: two parity disks

RAID 1


RAID 1
A1
A2
A3
A4
A1
A2
A3
A4
Disk 0
Disk 1


RAID 0+1
RAID 1
RAID 0
RAID 0
A1
A3
A5
A7
A2
A4
A6
A8
A1
A3
A5
A7
A2
A4
A6
A8
Disk 0
Disk 1
Disk 2
Disk 3


RAID 1+0
RAID 0
RAID 1
RAID 1
A1
A3
A5
A7
A1
A3
A5
A7
A2
A4
A6
A8
A2
A4
A6
A8
Disk 0
Disk 1
Disk 2
Disk 3

RAID 5

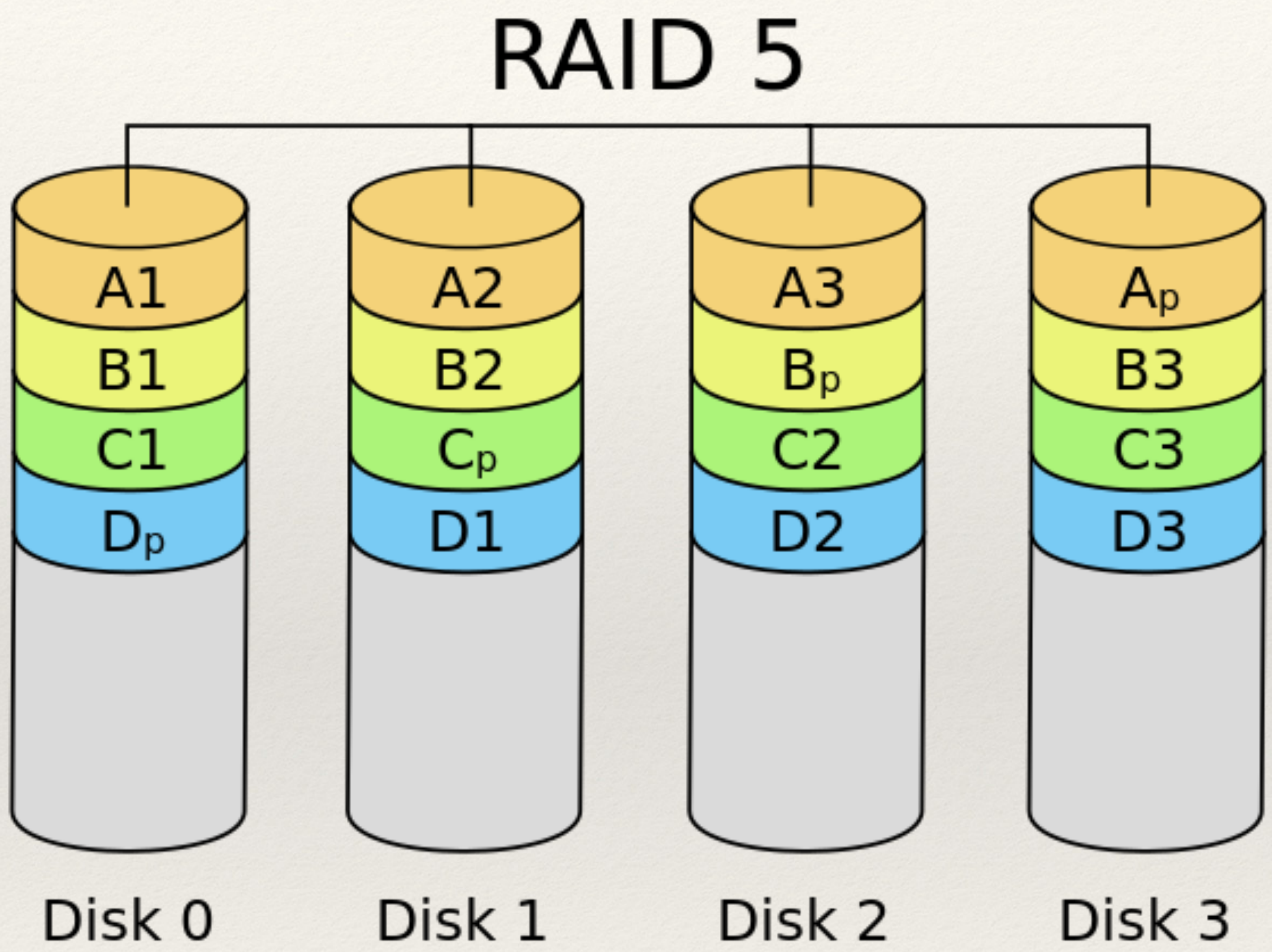
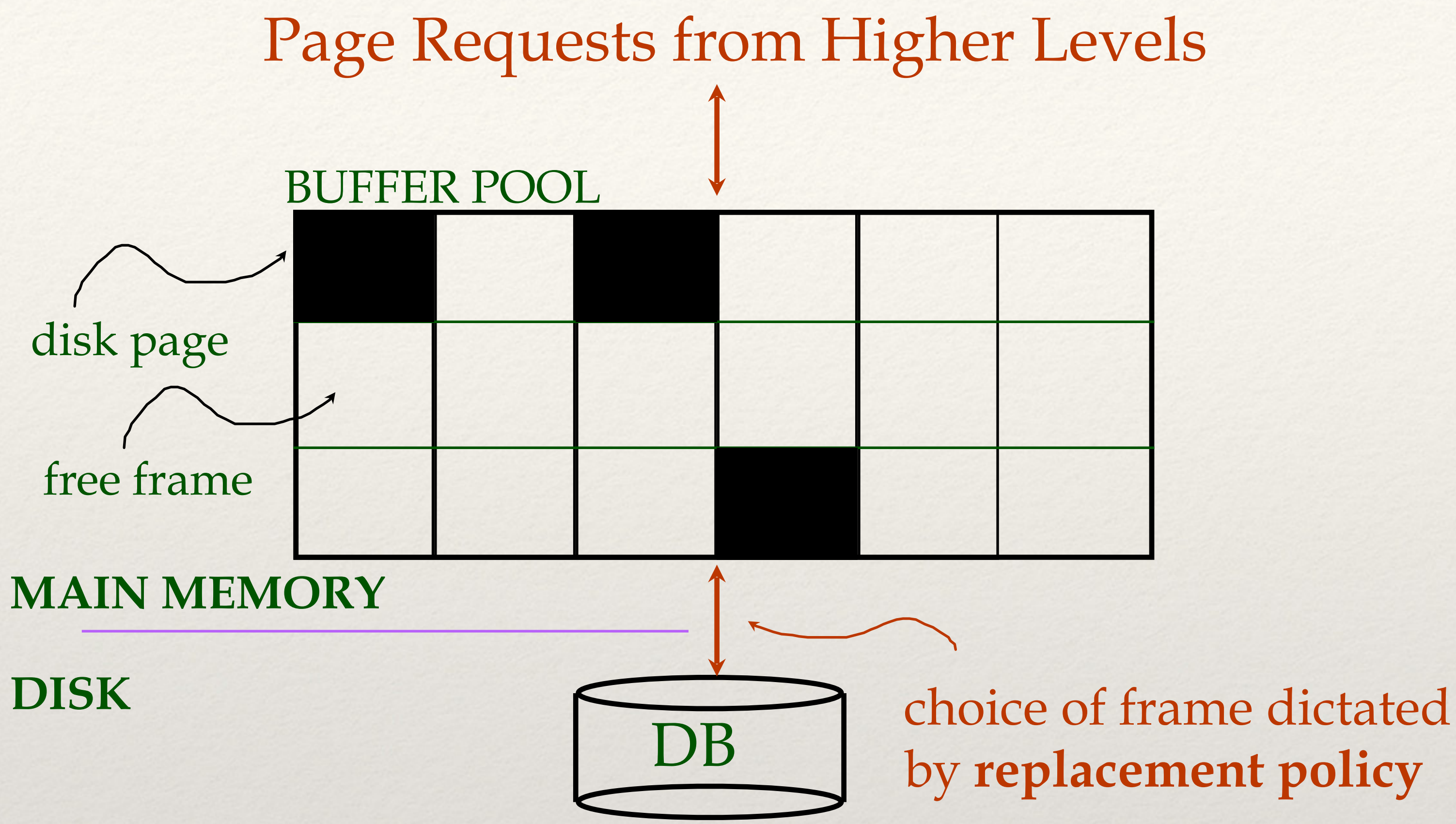


Diagram of a RAID 5 setup with distributed parity with each color representing the group of blocks in the respective parity block (a stripe).

Data in Memory

- Files organized as data blocks or pages
- Data must be in memory to operate on it
- Pages are brought to memory as needed
- The pages are “pinned” in memory while being used
- Buffer pool manager’s job to keep it well used

Buffer Management



- Schematic of a Buffer Pool
- Maintain the table of <frame #, pageid> + more information

Page Request

- Check if requested page is already in pool
- If not then
  - if pool is full, then choose a frame for replacement
  - if frame is "*dirty*", write it
  - read the requested page into the empty frame
- *Pin* the page and return it's address to the requestor
- Pages can be pre-fetched into buffer, e.g. for sequential scans

Page Request

- When done, the requestor of page must
  - *unpin* it
  - set the *dirty* bit to indicate if the page is modified
- A page may be requested by several requestors
  - keep track using a *pin count*
  - page is candidate for replacement iff *pin count* = 0
- Additional I/O may be needed for transaction management

Page Replacement

- When no more empty frames are available, a page must be replaced from the buffer pool
- Replacement policy guides the pick of suitable frames
  - LRU, clock, multi-policy
- Knowledge of access pattern can help guide the policy
  - E.g. data that's only usable once can be marked for immediate replacement
- Big memories help reduce the I/O as more data is cached

Why complex policies?

- Simple LRU is not sufficient in DBMS workloads
- Think about scanning a table larger than buffer pool more than once
  - Each page when needed again will not be in the pool (as it would have been “least recently” used and replaced)

Why LRU not ideal

- Scanning a three page (a,b,c) table twice
- 6 I/Os as nothing found in buffer
- with MRU scheme we have only 4 I/Os

Time	Frame 1	Frame 2	I/O
0			
1	a		1
2	*a*	b	1
3	c	*b*	1
4	*c*	a	1
5	b	*a*	1
6	b	c	1

Time	Frame 1	Frame 2	I/O
0			
1	a		1
2	a	*b*	1
3	a	*c*	1
4	*a*	c	0
5	*b*	c	1
6	b	*c*	0

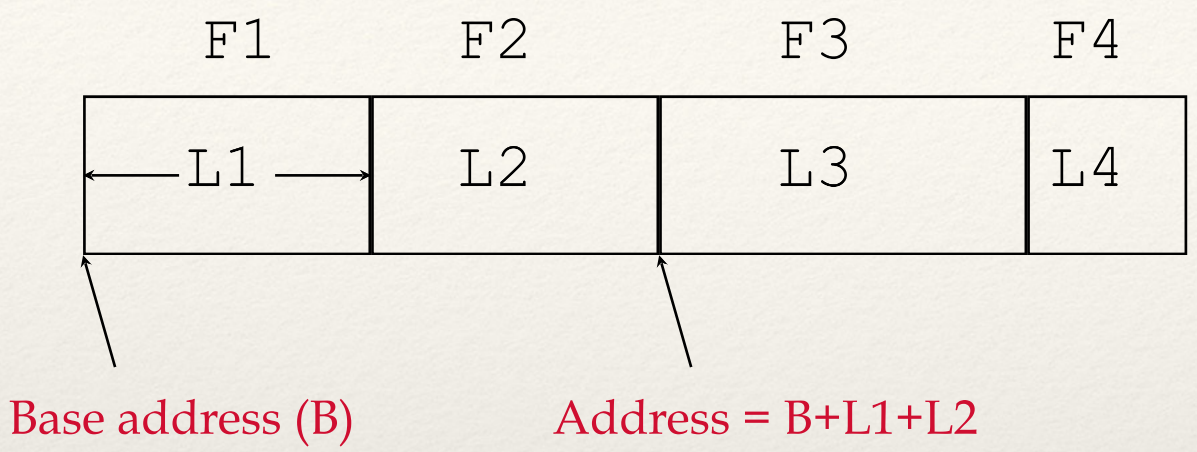
Why not just OS File System?

- The needs are more specialized
- Portability issues across different OS's
- Can't ensure optimal layout of files
- Some limitations, e.g. files can't span multiple disks
- Buffer management in DBMS requires ability to
  - pin a page in pool
  - force (i.e. write) a page to disk
  - adjust fetch and replacement policy as needed

What's on a Page?

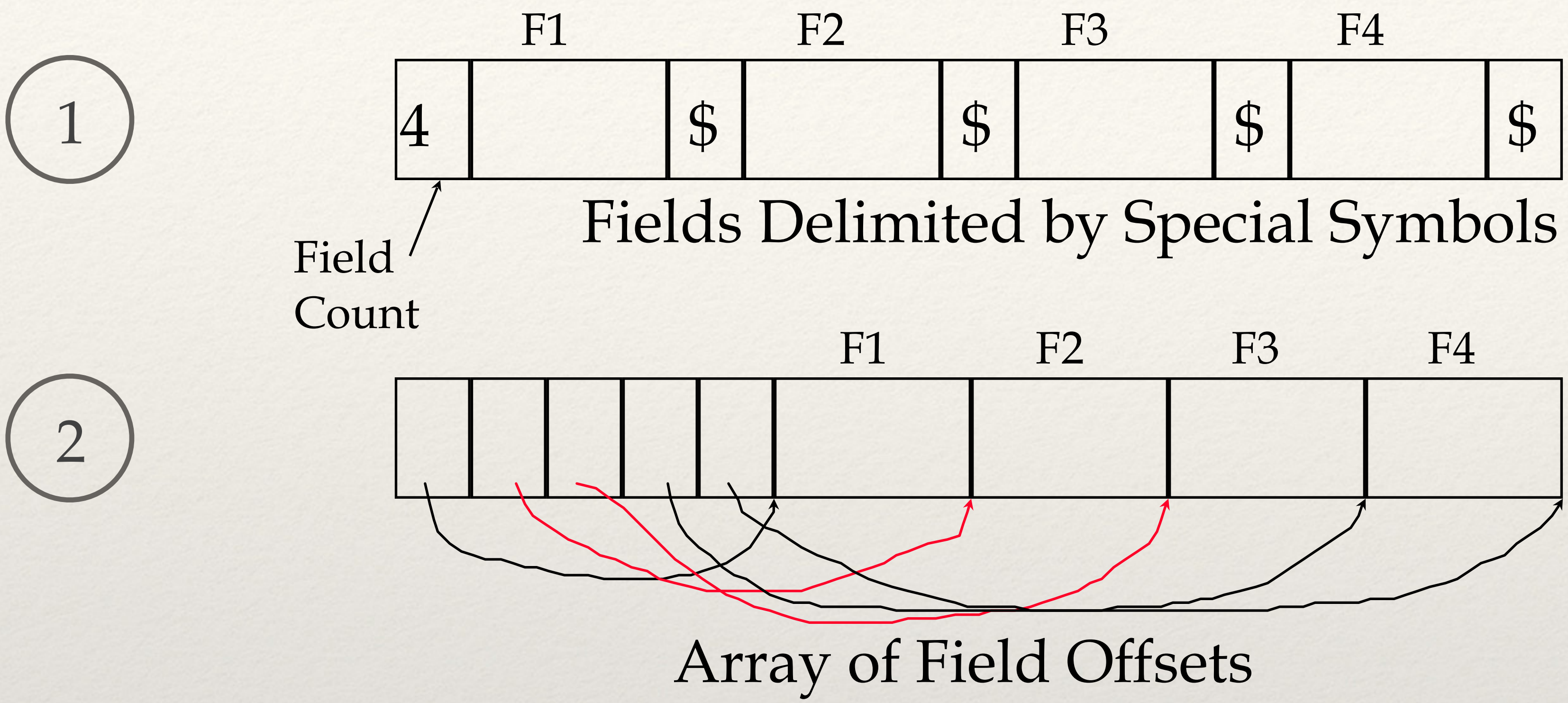
- Entire rows to just one column
- Rows can be fixed length (rare in a real DBMS)
- Rows are usually variable length, so a DBMS pages are designed for that
- Null Values
- Compressed values
- Compressed pages

Fixed Length Records



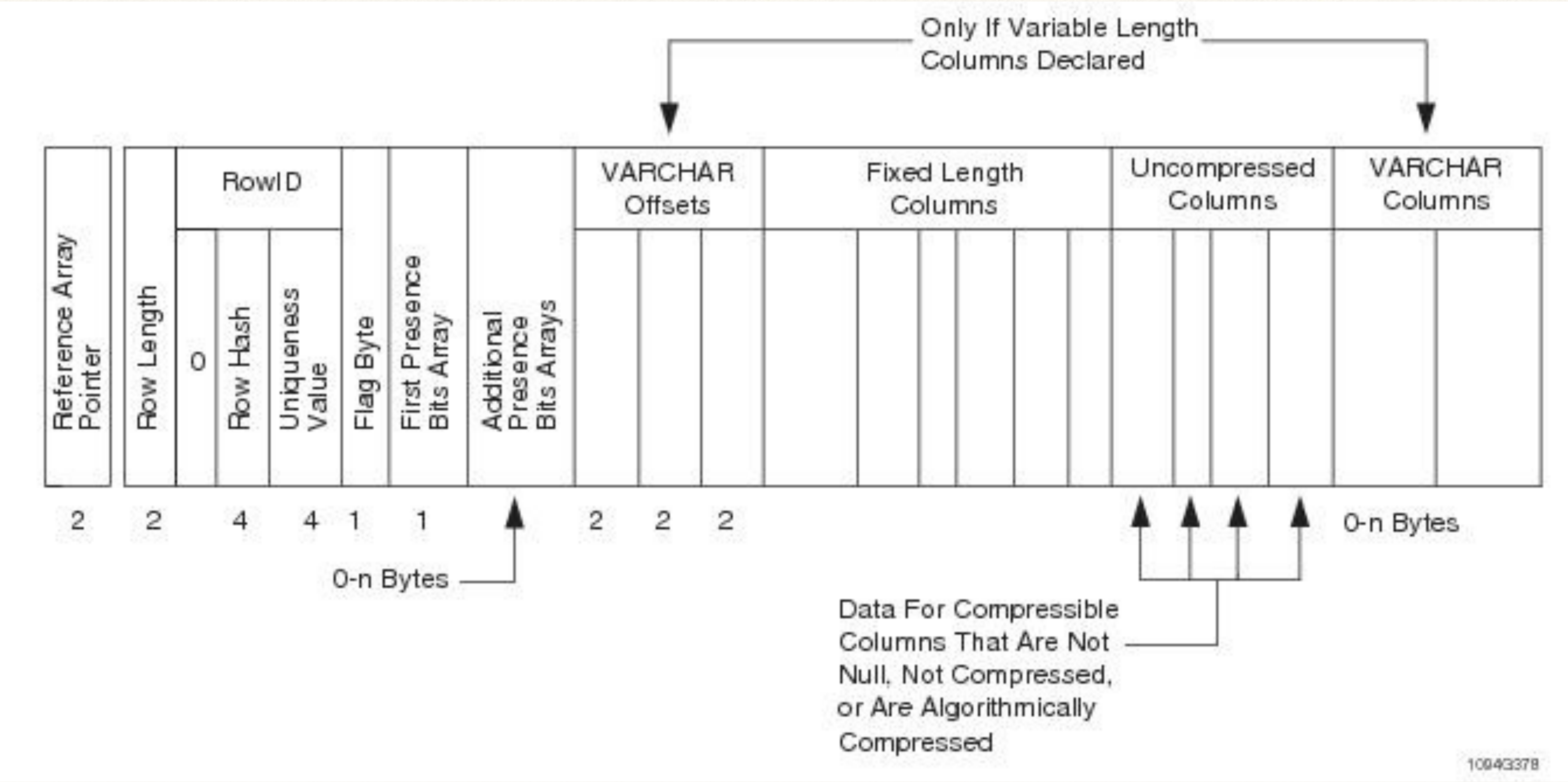
- Information about field types and sizes stored in system catalog
- Can simply look up i^{th} field, without reading the rest
- Same as an in-memory fixed length structure

Variable Length Records

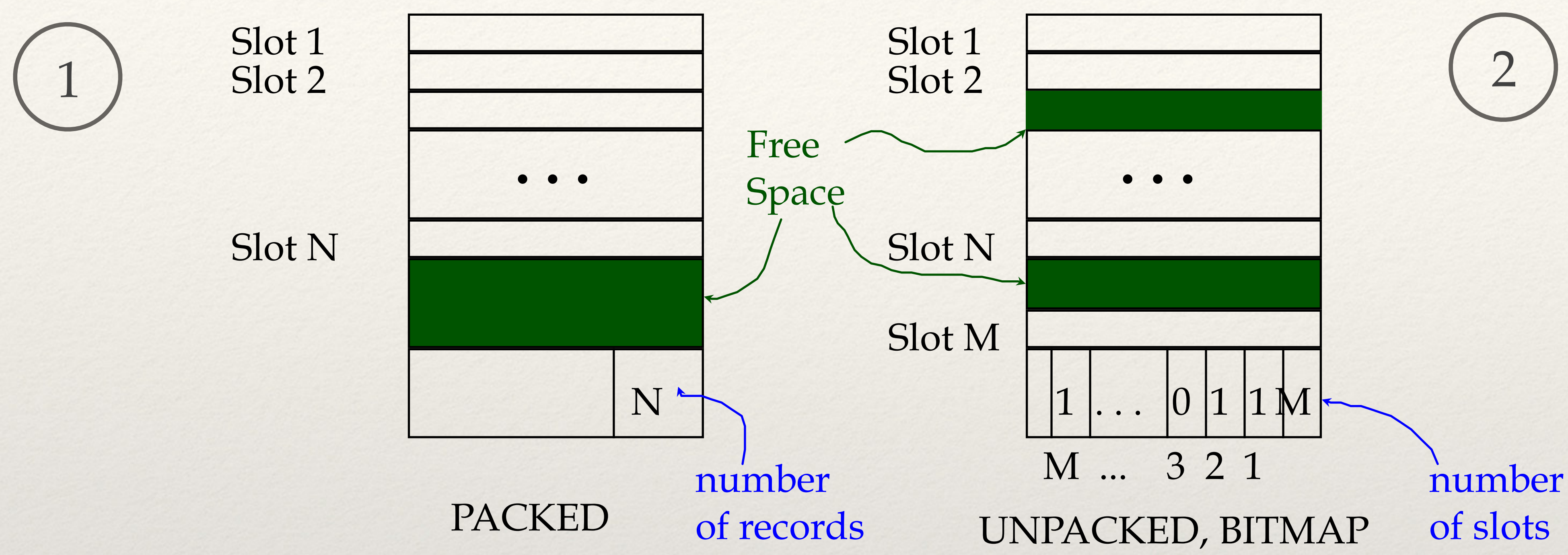


- Two alternative formats (#fields is fixed)
- Second offers direct access to i^{th} field, efficient NULL storage, and small overhead

Teradata Row Format

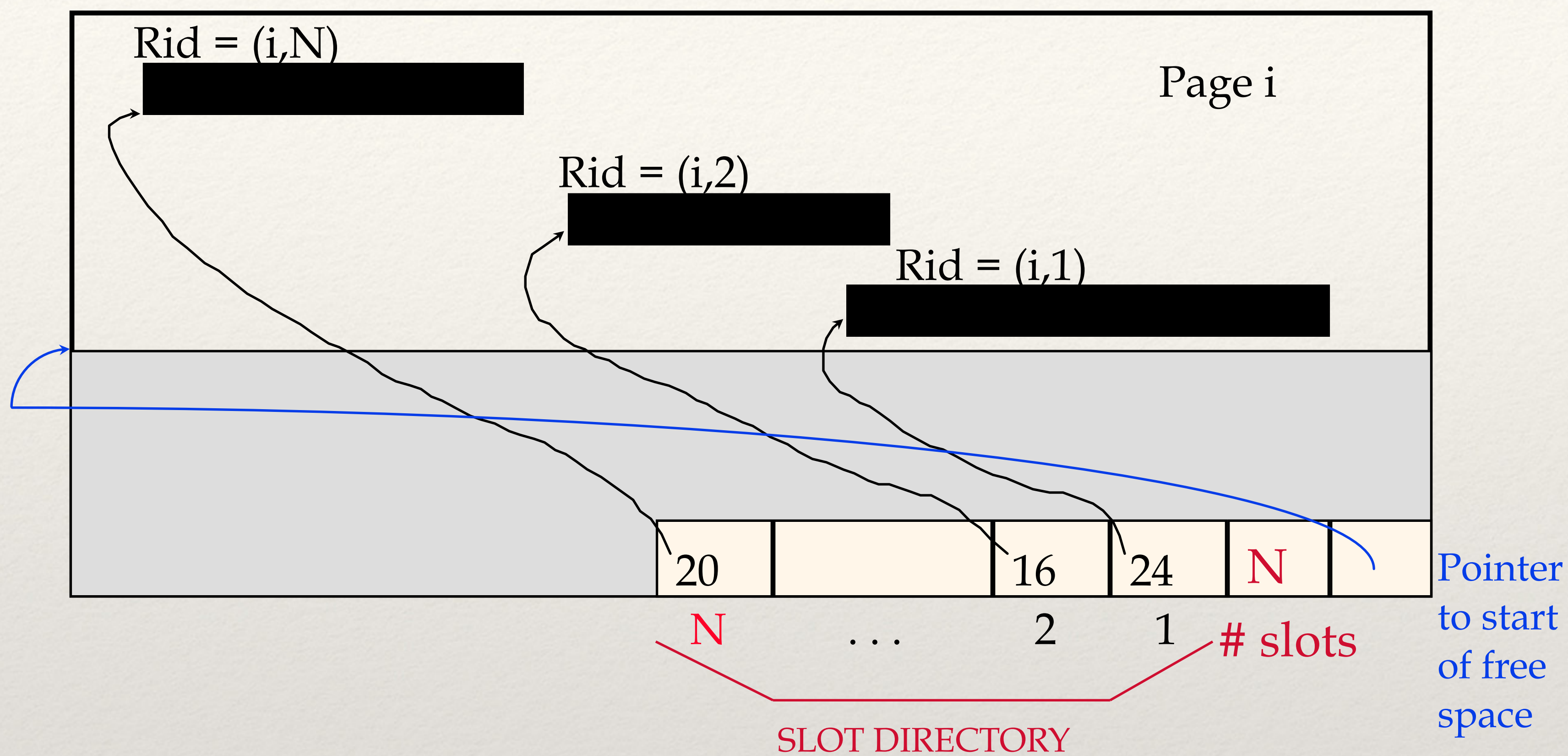


Page Format for Fixed Length Records



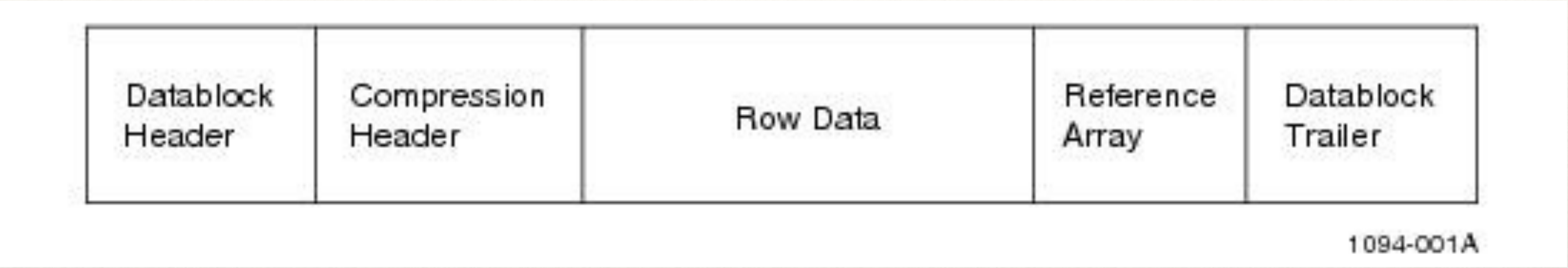
- *Record id (rid)* = <page id, slot #>
- (1): easy to insert; deletion requires compacting; moving records for free space management changes *rid* which may not be acceptable
- (2): find free slot to insert; no compaction after delete; no *rid* change

Page Format for Variable Length



- keep pointer to beginning of record (offset) and length in the slot directory
- need to to space management on page, bottom of page usually set for directory

Teradata Format



- Compression header for block level compression
- Header and trailer capture metadata about block

Data Compression

- Got to see a hint in the Teradata row/block formats
- Data in a table or a block can be compressed in several ways
- Dictionary encoding, common values represented as mere bits in the row (looked up at run time)
- Run length encoding: repeated consecutive values are kept once with a count (useful for ordered data)
- Block level compression (compress the whole data block)

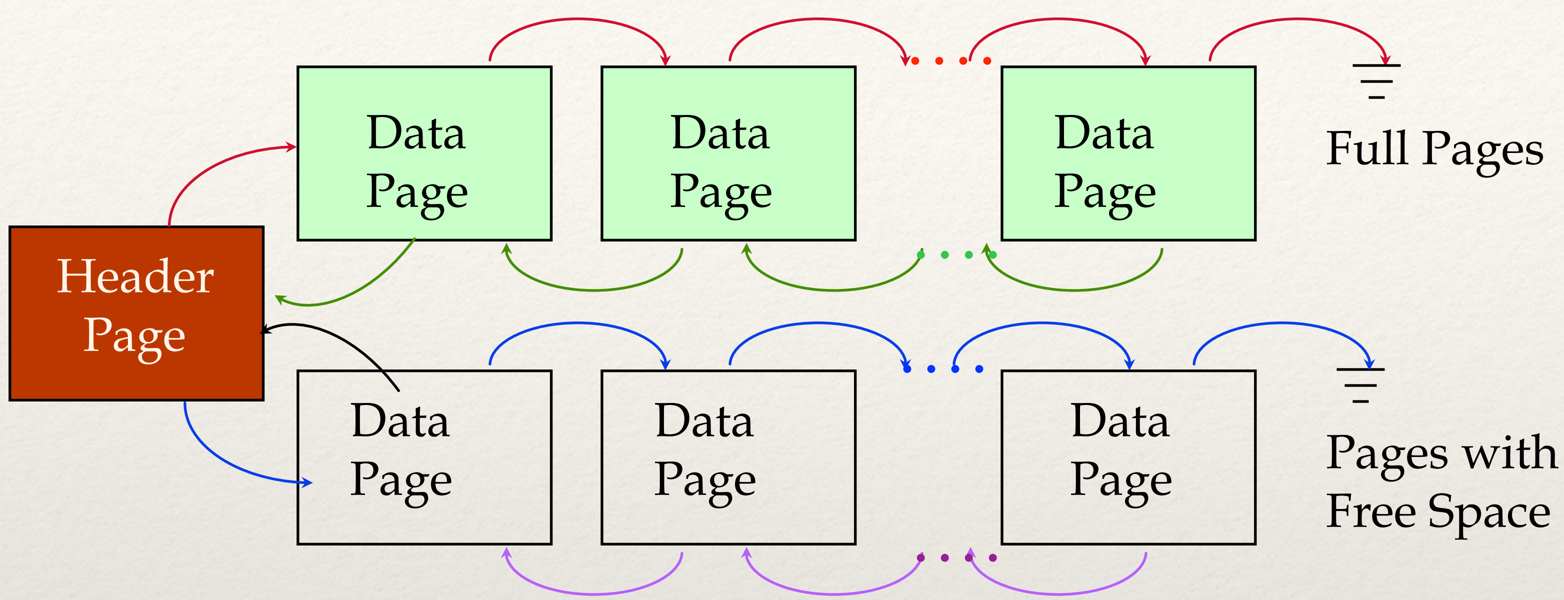
Why Data Compression?

- Disks are cheap so why?
- Performance: both dictionary and run length encoding save on I/O
- For HDD at least, since CPU’s are now much faster, even block level compression could help with matching I/O to CPU

Files

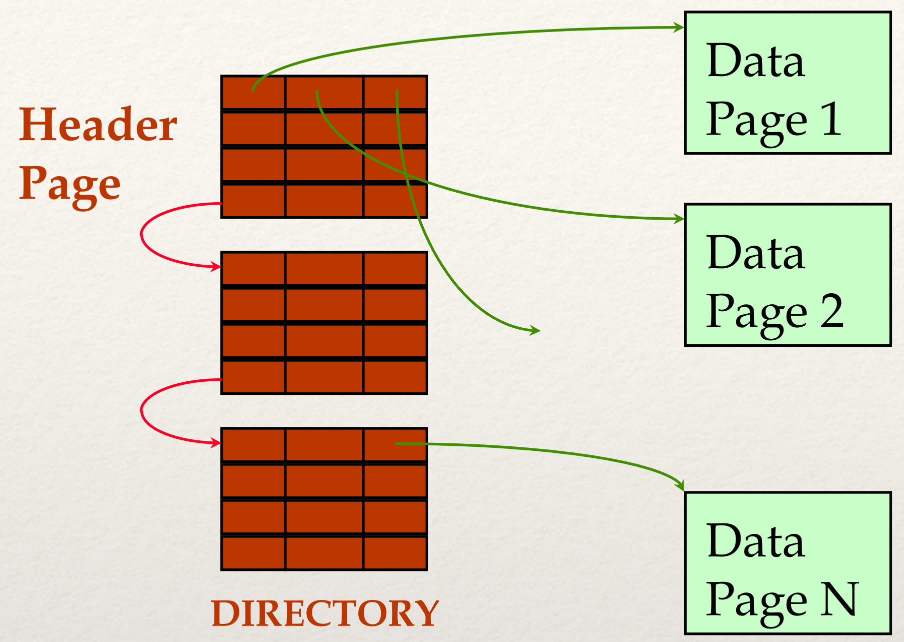
- Collection of pages containing related set of records (rows, or down to even a single column)
- Insert/delete/update the records
- Able to read a particular record via *rid* or
- Read all records or some records satisfying a condition

Unordered (Heap) Files as List



- Name and header page id is stored in a catalog
- Each page contains two pointers + data

Heap Files using Page Directory



- Directory entry for page can include free space information
- Directory is itself a collection of pages (usually very small)
- Finding a page with enough free space easy if maintained in the directory

Review

- Databases must have persistent storage: SSD, HDD
- Their performance characteristics affect database design
- Buffer manager tries to keep the optimal set of data blocks (pages) in memory to minimize I/O
  - must be able to “lock” (pin) a page in memory
  - must be able to write / flush page to disk on demand

Review

- Rows comprise both fixed and variable length fields
- Slotted page format is flexible to keep records organized and accessible on a page
- Single column values could be stored in fixed length records, but are usually compressed via encodings
- Compression is used at both column and block level
- Heap files are ok but most databases use B+ trees