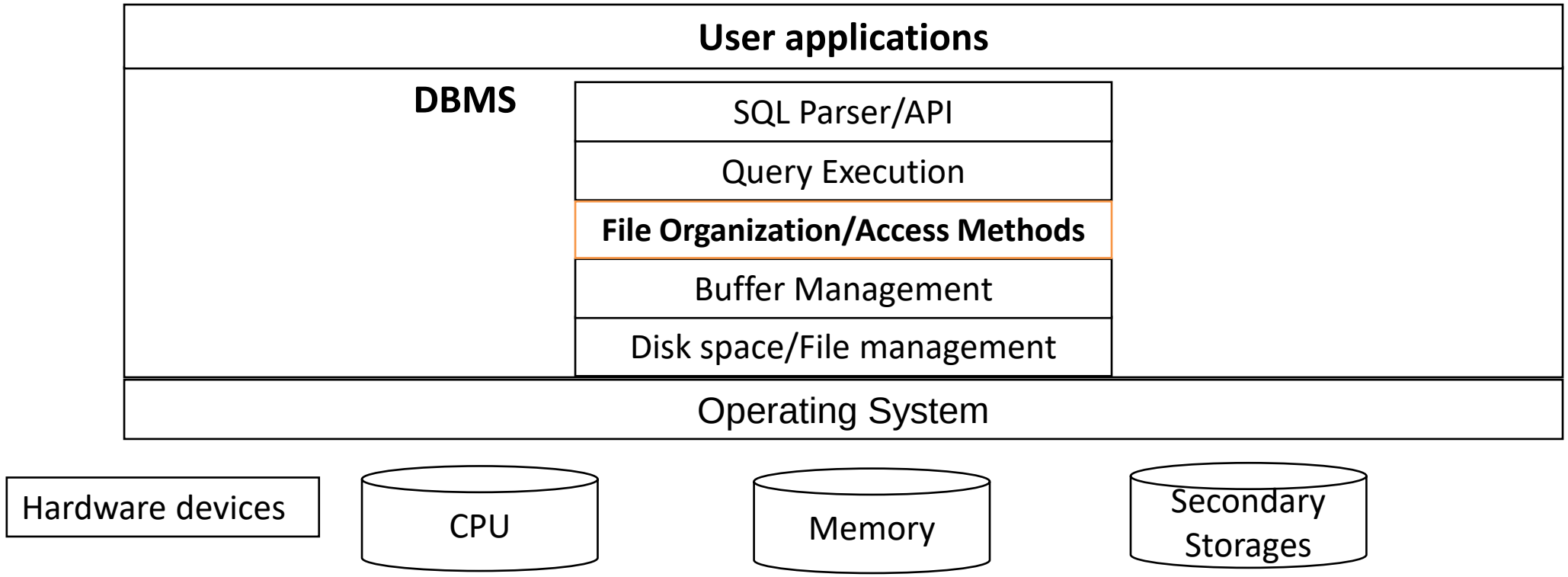
CSE462/562: Database Systems (Spring 22)

Lecture 6: File Organization and Access Methods

2/17/2022

University at Buffalo
Department of Computer Science and Engineering
School of Engineering and Applied Sciences

Big Picture

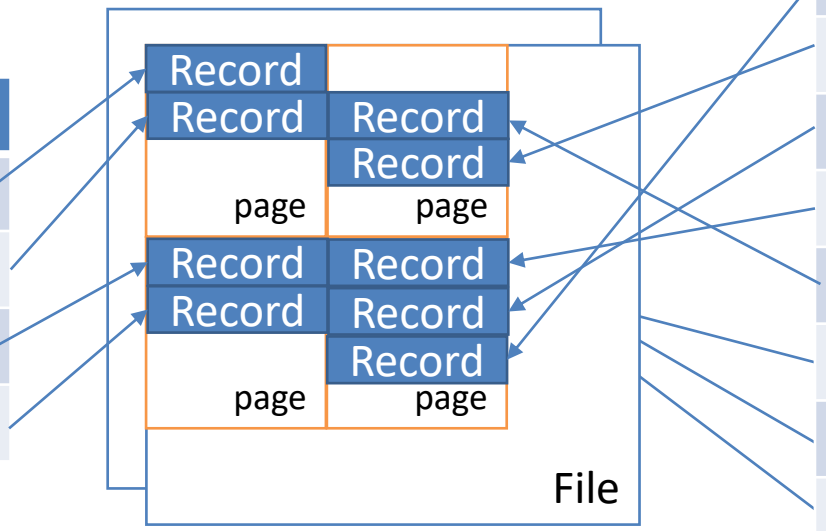

User applications
DBMS
SQL Parser/API
Query Execution
File Organization/Access Methods
Buffer Management
Disk space/File management
Operating System
Hardware devices
CPU
Memory
Secondary Storages

Database storage architecture

- Database is *logically* a collection of relations, which are (multi-)sets of records (rows)
- Physical mapping (in a traditional row-oriented DBMS)
  - Database -> files
  - Records -> contiguous bytes on fixed-size pages (e.g., 4KB)
    - Assumption: each record fits in a page
    - What if a record does not fit?

student

sid	name	login
100	Alice	alicer34
101	Bob	bob5
102	Charlie	charlie7
103	David	davel



enrollment

sid	semester	cno	grade
100	s22	562	2.0
102	s22	562	2.3
100	f21	560	3.7
101	s21	560	3.3
102	f21	560	4.0
103	s22	460	2.7
101	f21	560	3.3
103	f21	250	4.0

- What about relations?

  One/several file(s) per relation? Mixing records from correlated relations in one/several file(s)?

Database catalog

- Catalogs are DBMS defined relations that
  - stores meta-information about
    - Relation schemas
    - Physical storage format and location
    - And many other important internal states
- *Can be implemented as regular relations*

Table

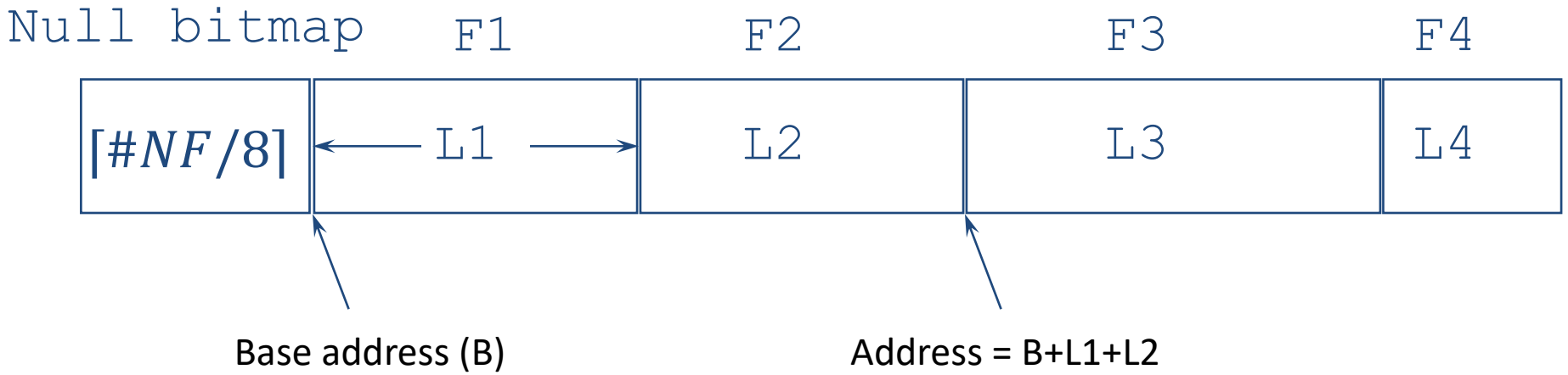
TABID	TABNAME	TABFPATH
1	TABLE	/dbdata/1
2	COLUMN	/dbdata/2
100	STUDENT	/dbdata/100
101	ENROLLMENT	/dbdata/101

Column

TABID	COLID	COLNAME	COLTYPNAME
1	0	TABID	OID
1	1	TABNAME	VARCHAR(64)
1	2	TABFPATH	VARCHAR(256)
2	0	TABID	OID
2	1	COLID	INT2
2	2	COLNAME	VARCHAR(64)
2	3	COLTYPNAME	VARCHAR(64)
100	0	SID	SERIAL
100	1	NAME	VARCHAR(32)
100	2	LOGIN	VARCHAR(40)
101	0	SID	INTEGER
101	1	SEMESTER	CHAR(3)
101	2	CNO	INTEGER
101	3	GRADE	DOUBLE

Record format: fixed-length

- Fixed-length record
  - Assuming all fields F1, F2, F3, … have known (maximum) length
  - Base address B: may be a file offset or a memory address
  - Lengths L1, L2, L3, … can be found using system catalog

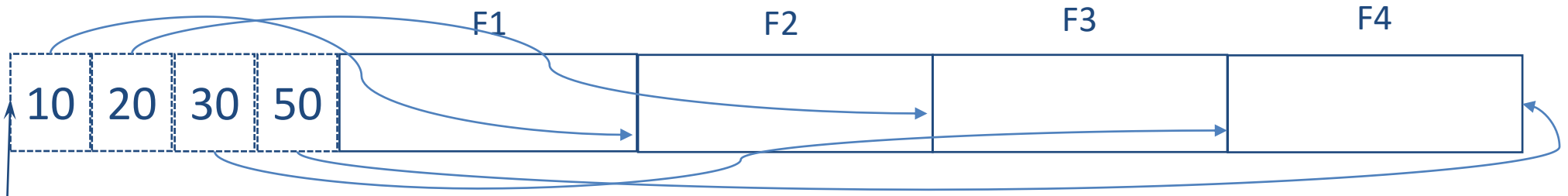


#NF: number of nullable fields

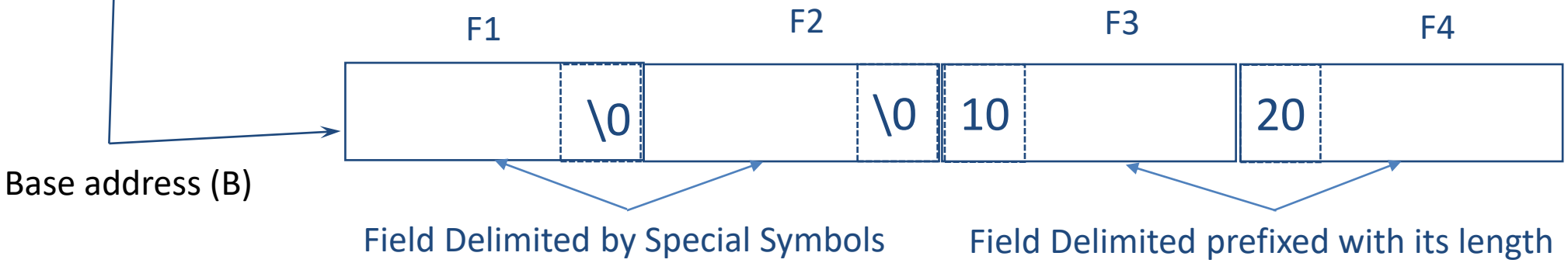
- How to handle NULL values?
  - *Null bitmap*
- Is this practical? How to deal with variable-length types (consider `VARCHAR(n)`)?
  - Waste of space (e.g., names are rarely very long, but we must reserve space for the longest name)

Record format: variable-length

- Variable-length record
  - Two main approaches:
    - Using offset array
      - random access to fields given B, but takes more space



    - Using self-contained data field (with separator/encoded offset)
      - Computed offsets (e.g., offset of F3 = L1 + L2); but is usually more compact



- Handling NULLs? Null bitmap!
- Many possible designs with minor tweaks for different space/time efficiency trade-offs
  - Can also combine both fixed-length and variable-length record formats

Data alignment in records

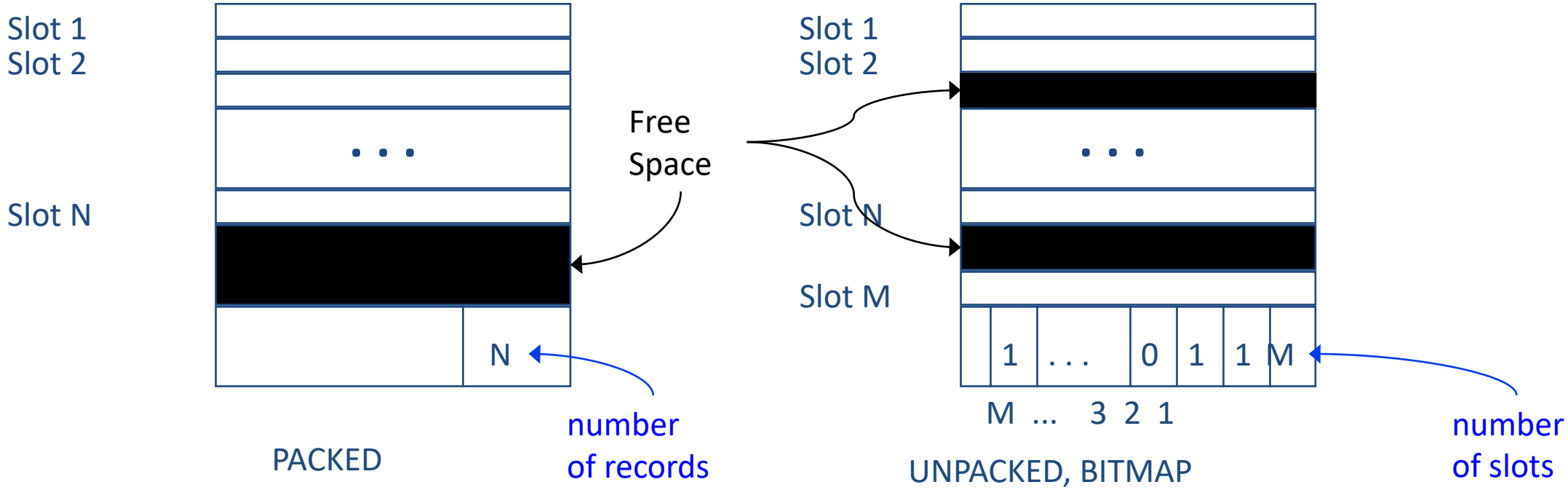
- Alignment requirements?
  - Alignment example: to read/write a 32-bit integer, it should be at $some\ address\ mod\ 4\ ==\ 0$
  - Most architecture has alignment requirements
    - Some requires alignment (most RISC arch, e.g., ARM v5 or earlier)
    - Some don't but have restrictions/performance loss/atomicity issues (e.g., x86_64/newer ARM)
  - By default, compilers automatically align values properly that for you
  - DB records? Two choices:
    - Pack everything, and memcpy the field before access
      - Less efficient, but save space
    - Align offsets manually
      - More efficient field access, but waste space

```
struct A {
  int32_t x;
  int16_t y;
  int64_t z;
};

// alignof(A) == 8
// offsetof(A, x) == 0
// offsetof(A, y) == 4
// offsetof(A, z) == 8 (not 6!)
```

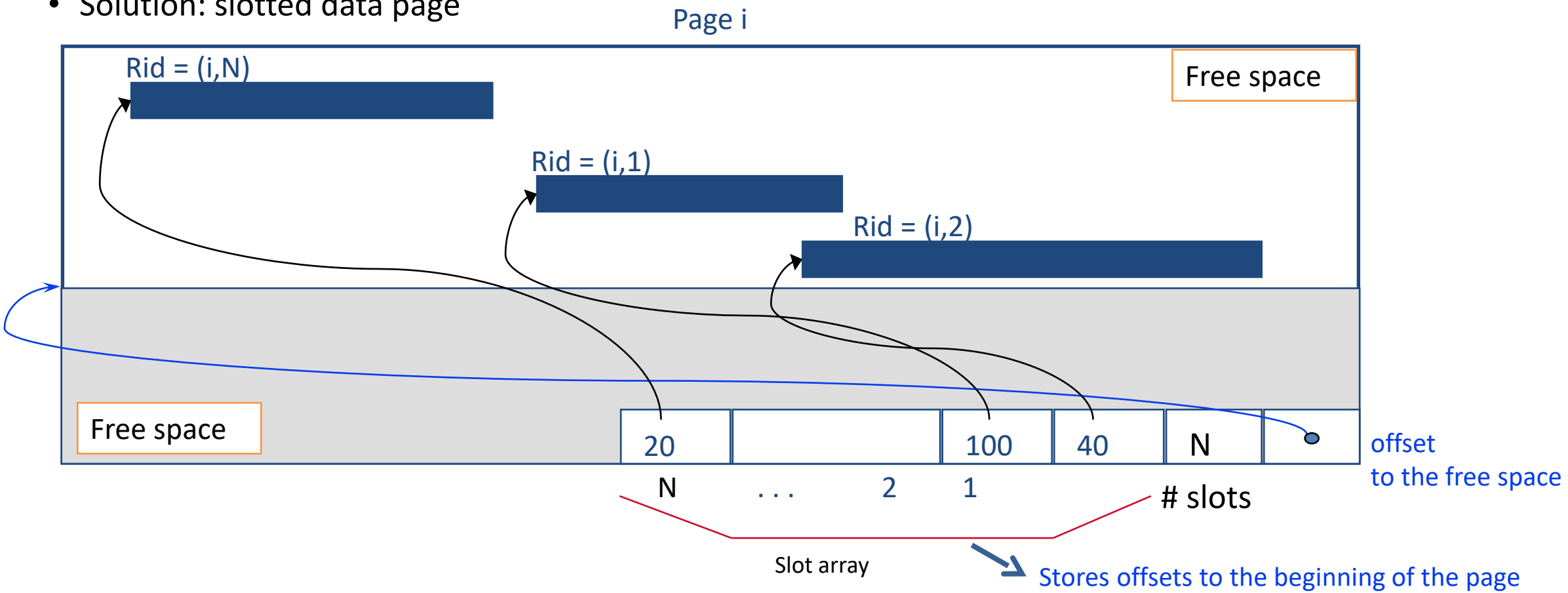
Page layout for fixed-length records

- Why not storing record consecutively in a file?
  - Hard to update/delete!
- How do we store records in fixed-size pages?
  - Fixed-length record: easy
    - Not usually used as it wastes space



Page layout for variable-length records

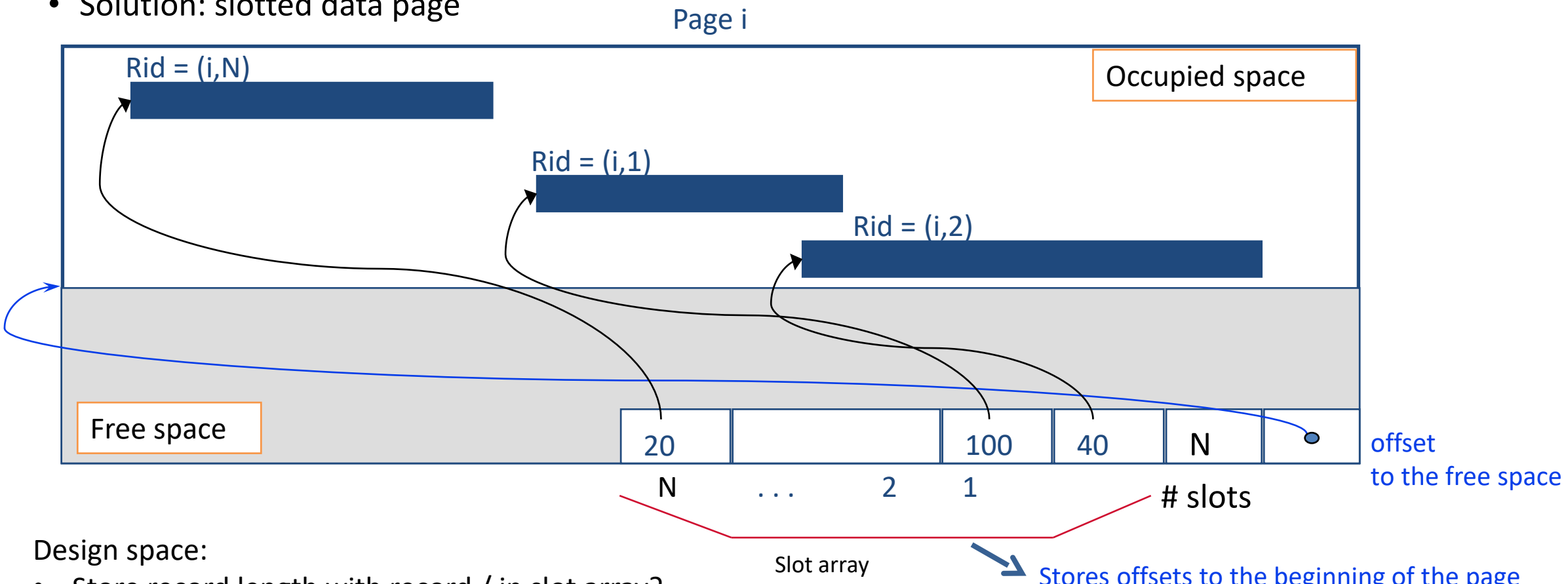
- What about variable-length records?
  - Solution: slotted data page



Can move records within a page without changing its record id.

Page layout for variable-length record

- What about variable-length records?
  - Solution: slotted data page



Design space:

- Store record length with record / in slot array?
- Allow free space within the occupied space?
  - Eager vs lazy compaction?
- Optional page header?

Organizing pages in a heap file

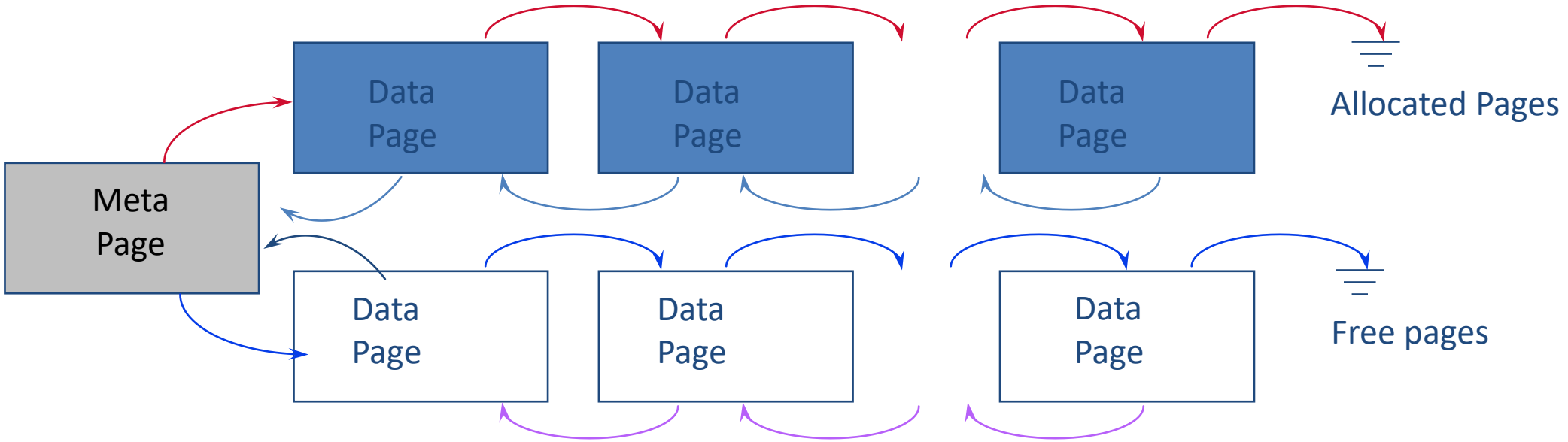
- Heap file is the most basic and common way of managing pages for a single relation
  - Consists of a collection of fixed-size pages
  - Pages/records are unordered
- Heap files must support
  - Efficient insertion/deletion/update of records
  - Efficient access of a record
  - Efficient enumeration of all the records
  - Management of free space (also managed by disk space manager/file system)

- Note
  - A heap file does not necessarily map to a single file on FS
    - A heap file can span multiple FS files (e.g., PostgreSQL)
  - A file on FS does not necessarily only store pages for a single heap file
    - All heap files are stored in a single FS File (i.e., single-file DBMS such as SQLite)
    - Our course project Taco-DB: stores pages of different heap files across a number of files on FS

Organizing pages in a heap file

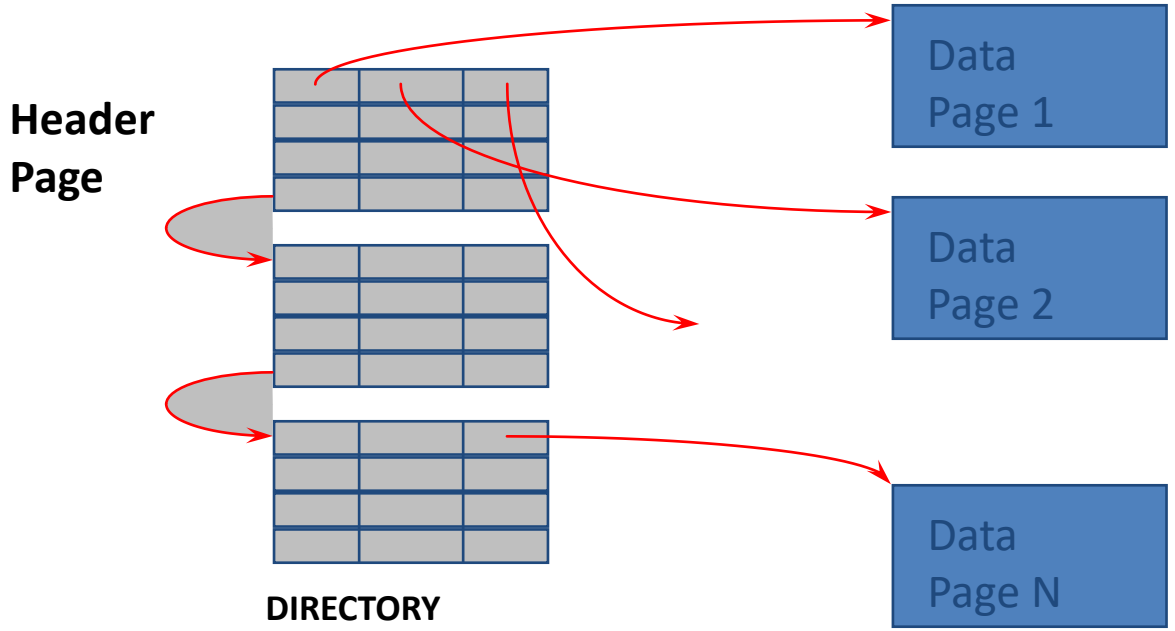
- Many possible alternatives and variants
  - We consider the most representative two of them

Heap file alternative 1: doubly-linked lists



- The header page id and Heap file name must be stored someplace.
  - Database catalog
- Each page contains 2 `pointers' plus data.
  - What are these pointers? Page Number and/or File ID?
- Supports sequential access
  - Random access? Only if you know the page number (and the underlying file system supports random seek)
- Does enumerating the pages through the next pointers always incur sequential I/O?
  - Not necessarily! Depending on how you allocate pages.

Heap file alternative 2: page directory



- The entry for a page can include the number of free bytes on the page.
  - Or use free space bitmap on a contiguous space.
- The directory is a collection of pages; linked list implementation is just one alternative.
  - Can also allocate contiguous pages for page directory for faster random access and/or using hierarchical page directory
  - *PD is much smaller than the all data pages*!

Revisit of the big picture of file organization

- Fields → Records → Pages → Heap Files (→ Files on File System) → Storage Device

- What we support:
  - Insert a record -- easy
  - Update/delete of a record with known record ID
  - Enumerating all data records

  - How do I find the student with name “Alice”?
    - Linear search for record ID
  - Can we do better?
    - Binary search? Binary tree? B-tree? Hash table? Partitioning
    - Do we always need to store records as a whole?
    - Needs alternative file organization

- Sometimes also called access methods (a name comes from mainframe OS)
  - data structures and algorithms for sequentially or randomly retrieving data by keys

Alternative file organization

- Heap file: unordered, good for enumerating all records
- Sorted file: best for random retrieval by *search key* and/or in search key order
  - A *search key* is a set of attributes (can be a single attribute) that the underlying file is sorted w.r.t.
    - Has nothing to do with (primary/candidate) keys!
  - Must be based on files that support random access of data pages with consecutive page numbers
    - e.g., for a sorted file with M pages, page numbers are 0, 1, 2, …, M-1.
    - Need support for random access of page i efficiently without linear traversal

- Compare the costs of record insertion/deletion/search in sorted file vs heap file?

Cost Model for Analysis

- We assume fixed-length records and ignore CPU costs, for simplicity:
  - N: the number of records
  - B: Number of records per page
  - T: Number of matching record in a search
  - Cost model: # of I/Os (also ignoring pre-fetching and/or random vs sequential access), and thus even I/O cost is loosely approximated.
  - Average-case analysis (unless o/w specified); based on several simplistic assumptions.
    - Good enough for knowing the overall trends.
    - Reality is a lot messier than this.
- Additional assumptions
  - Single record to insert and delete; unless o/w specified
  - Equality selection - exactly one match; unless o/w specified
  - Heap Files:
    - Insert always appends to end of file.
  - Sorted Files:
    - Two alternatives:
      - No need to compact the file after deletions.
      - Files compacted after deletions.
    - Selections on search key (the attribute(s) used for sorting).

Cost of operations

N: Number of records
N/B: The number of data pages
B: Number of records per page
T: Number of matching records

# of I/Os	Heap File	Sorted File
Scan all records	N/B	N/B
Equality Search: if we know there's only 1 matching record	0.5N/B, Best Case: 1, Worst Case: N/B	$\log_2$ (N/B) Best case: 1.
Range Search	N/B	$\log_2$ (N/B) + #match pages = $\log_2$ (N/B) + T/B
Insert: compact for sorted file	2	$\log_2$ (N/B) + N/B (read + write for 0.5N/B pages on average)
Delete: no compact for sorted file	0.5N/B + 1	$\log_2$ (N/B) + 1

Cost of operations

N: Number of records
N/B: The number of data pages
B: Number of records per page
T: Number of matching records

# of I/Os	Heap File	Sorted File
Scan all records	N/B	N/B
Equality Search: if we know there's only 1 matching record	0.5N/B, Best Case: 1, Worst Case: N/B	$\log_2$ (N/B) Best case: 1.
Range Search	N/B	$\log_2$ (N/B) + #match pages = $\log_2$ (N/B) + T/B
Insert: compact for sorted file	2	$\log_2$ (N/B) + N/B (read + write for 0.5N/B pages on average)
Delete: compact for sorted file	0.5N/B + 1	$\log_2$ (N/B) + N/B

Alternative file organization (others)

- Heap file: unordered, good for enumerating all records
- Sorted file: best for random retrieval by *search key* and/or in search key order
  - A *search key* is a set of attributes (can be a single attribute) that the underlying file is sorted w.r.t.
    - Has nothing to do with (primary/candidate) keys!

- Columnar store: store individual column/column sets in separate files
  - Also called vertical partitioning
  - Good for queries with projection -- saves I/O, SIMD friendly

- Index files (next lecture)
  - Clustered index vs unclustered index
  - Primary index vs secondary index

Summary

- This lecture
  - Data storage layout
  - File organization
  - Cost analysis of heap file and sorted file access methods

- Next time:
  - Index files

- HW1 solution released today
- Project 2 released today
  - Due on 3/8/2022, 11:59 pm EST; start early and read the project page!
  - Write-up due on 3/10/2022, 11:59 pm EST.