CSE462/562: Database Systems (Fall 24)

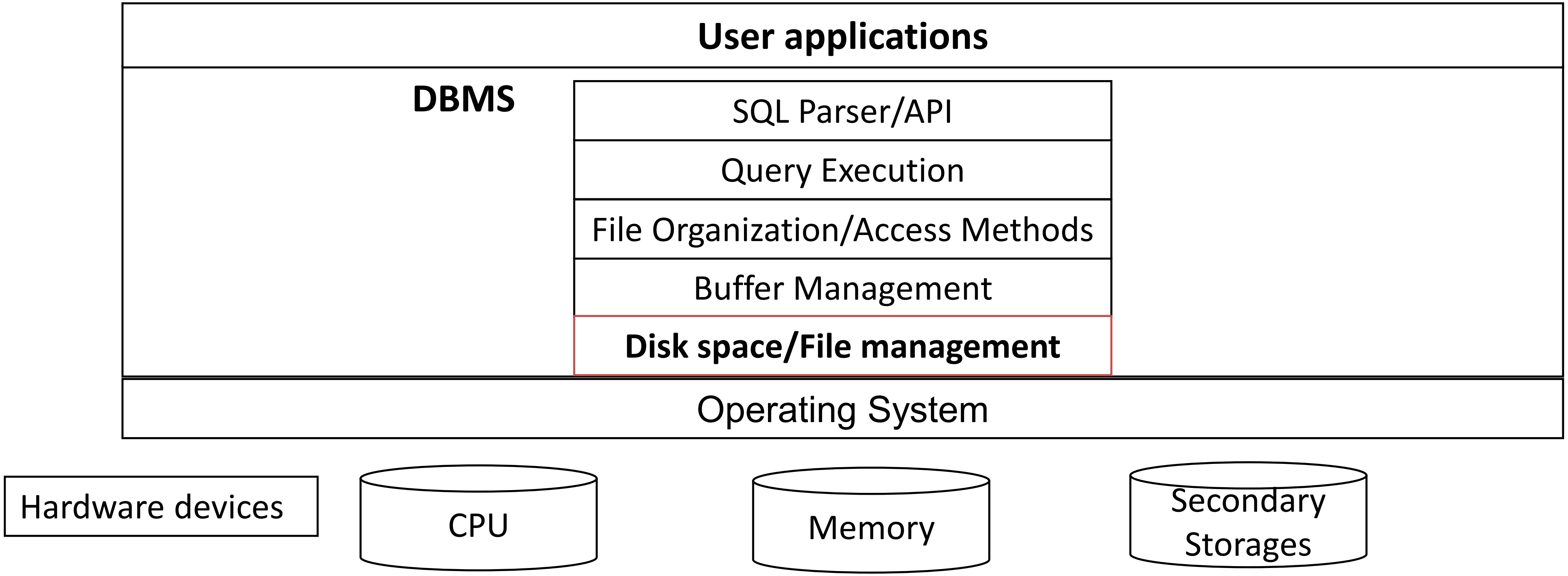
Lecture 3: Data Storage Layout

9/5/2024

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

Last Update: 9/5/24, 4:10 PM

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

Disk Space Management

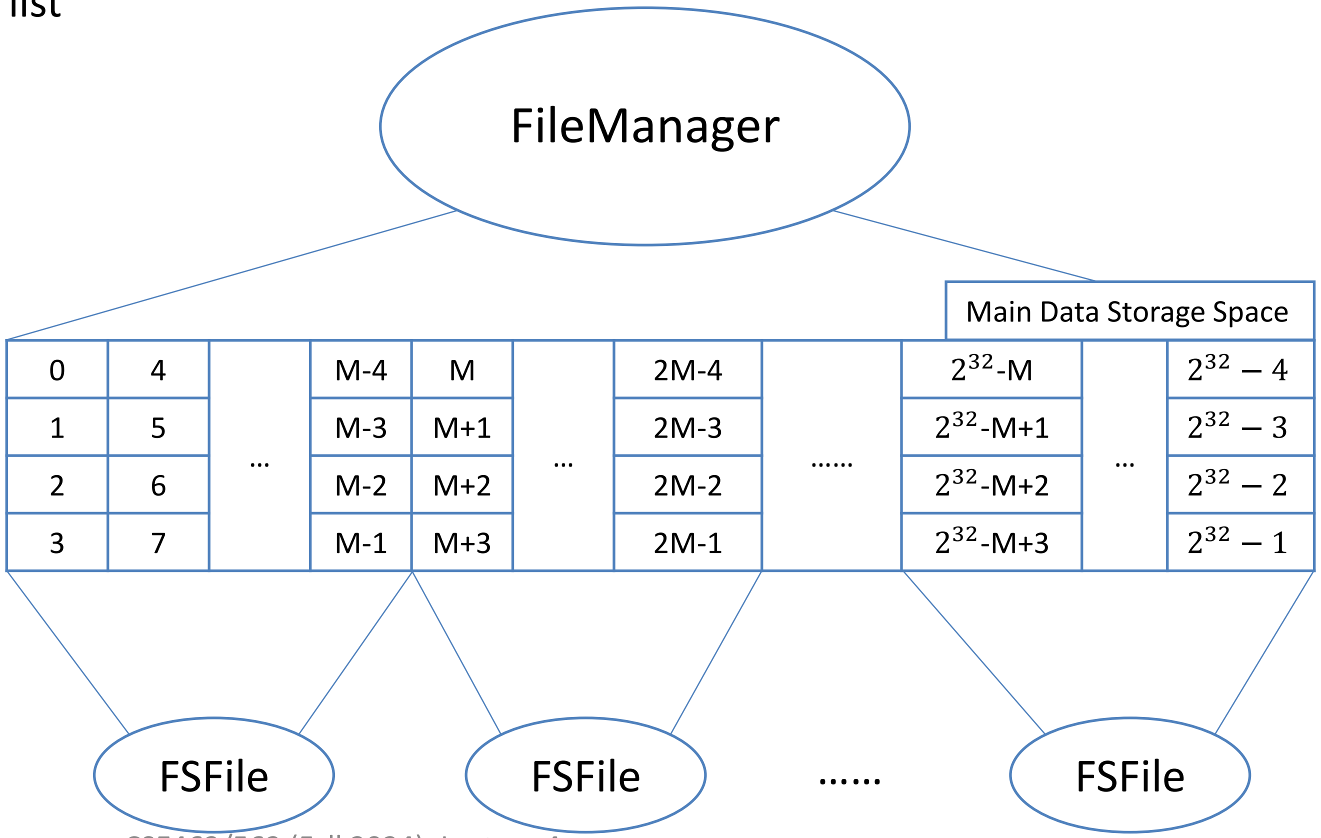
- Lowest layer of DBMS software manages space on disk
  - Disk space is usually organized in *pages*
    - which may not necessarily directly be mapped to disk sectors/file system pages!
    - common choices are 4KB, 8KB, 16KB, etc.
  - Using the OS file system or not? Some do and some don't!
  - Even with file system
    - How to organize pages (in one file/multiple files)?
    - How to deal with concurrency/recovery?
    - …

- Higher levels call upon this layer to:
  - allocate/de-allocate a page
  - read/write a page

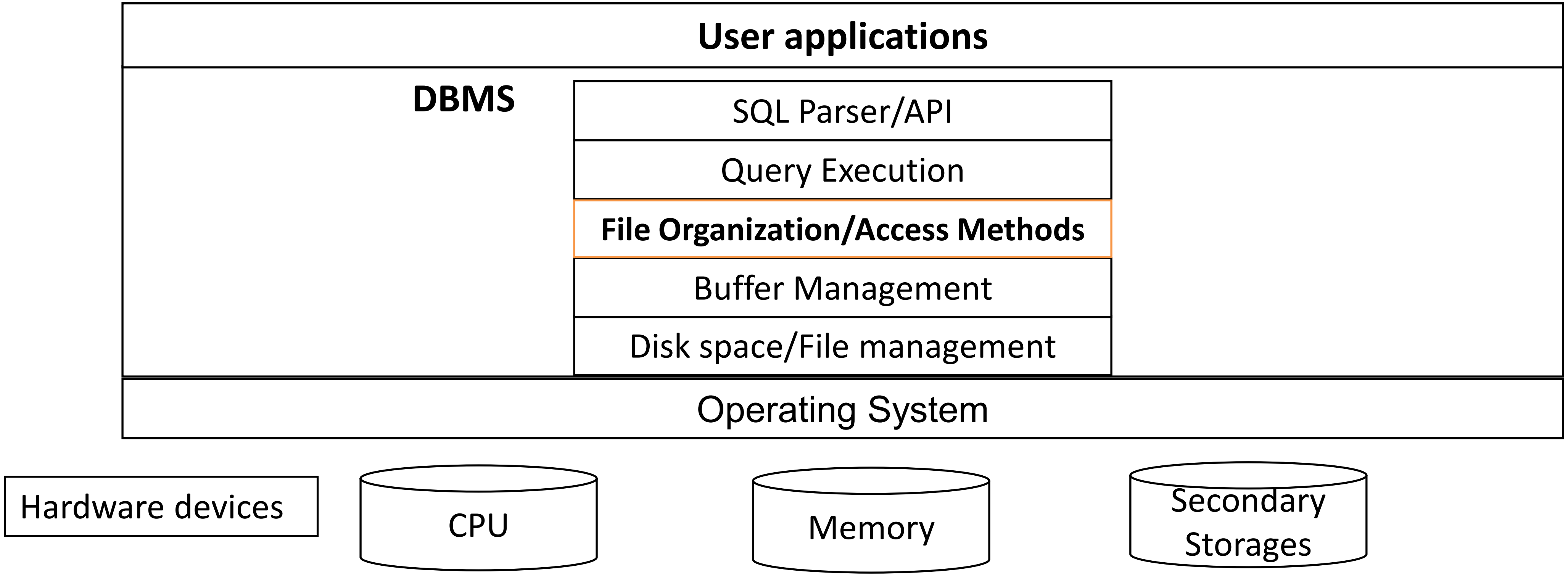
- Best if a request for a sequence of pages is satisfied by pages stored sequentially on disk!
  - Responsibility of disk space manager.
  - Higher levels don't know how this is done, or how free space is managed.
  - Though they may assume sequential access for files!
    - Hence, disk space manager should do a decent job.

Disk Space Management in course project Taco-DB

- A flat main data storage page from page 0 to page $2^{32} - 1$
  - Stored as 64GB files on the local file system;
  - One instance of FSFile manage a real file in the file system (e.g., allocate/read/write a page).
    - This is your task in Project 1 – lab 1.
  - FileManager manages many virtual files *(more on this next week)*
    - Each is a double-linked list of pages, allocated in groups of 64 consecutive pages
    - Each file maintains its own free list



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

Relational database

- A relational Database is *logically* a collection of *tables* (aka *relations*)
- **Table schema**: each *table* has one or more *fields* (aka *columns*)
  - Each *field* has a type and (usually) a name
- **Table instance**: a *table* is a (multi-)set of *records* (aka *rows/tuples*)
  - Each *record* has one value or NULL for each *field* in the *table schema*
    - The field type dictates the set of valid values

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

student

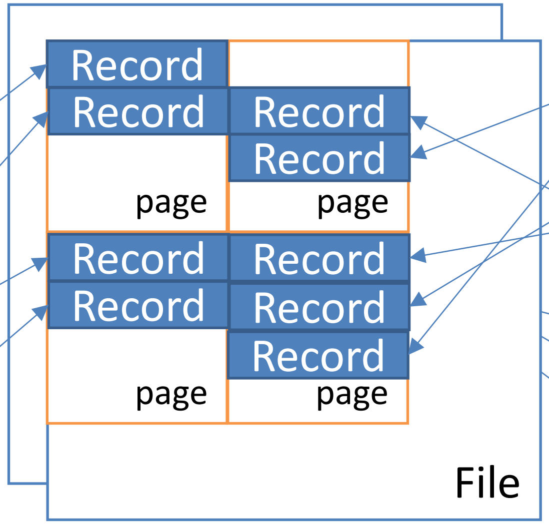
sid	name	login
100	Alice	alicer34
101	Bob	bob5
102	Charlie	charlie7
103	David	davel

Database storage architecture

- Mapping from relational database to physical storage
  - 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



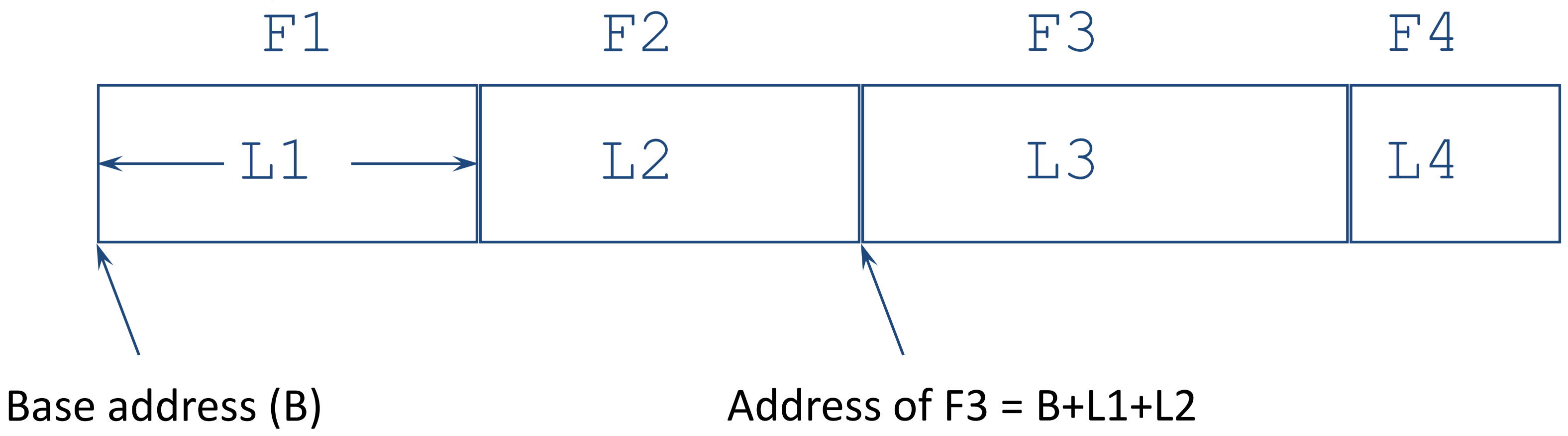
enrollment

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

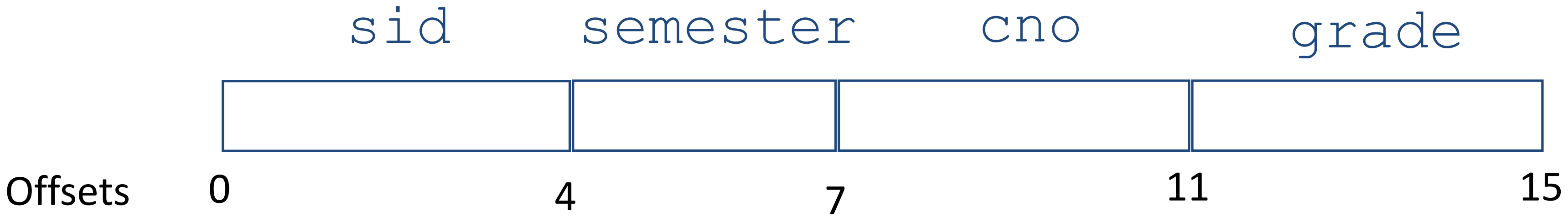
- What about relations?
  One/several file(s) per relation? Mixing records from correlated relations in one/several file(s)?

Record format: fixed-length

- Fixed-length record
  - Assuming all fields F1, F2, F3, … have known (maximum) length
    - Denote the maximum lengths as L1, L2, L3, …
  - Base address B: may be a file offset or a memory address
  - Offset of field $Fi = \sum_{j=1}^{i-1} Li$

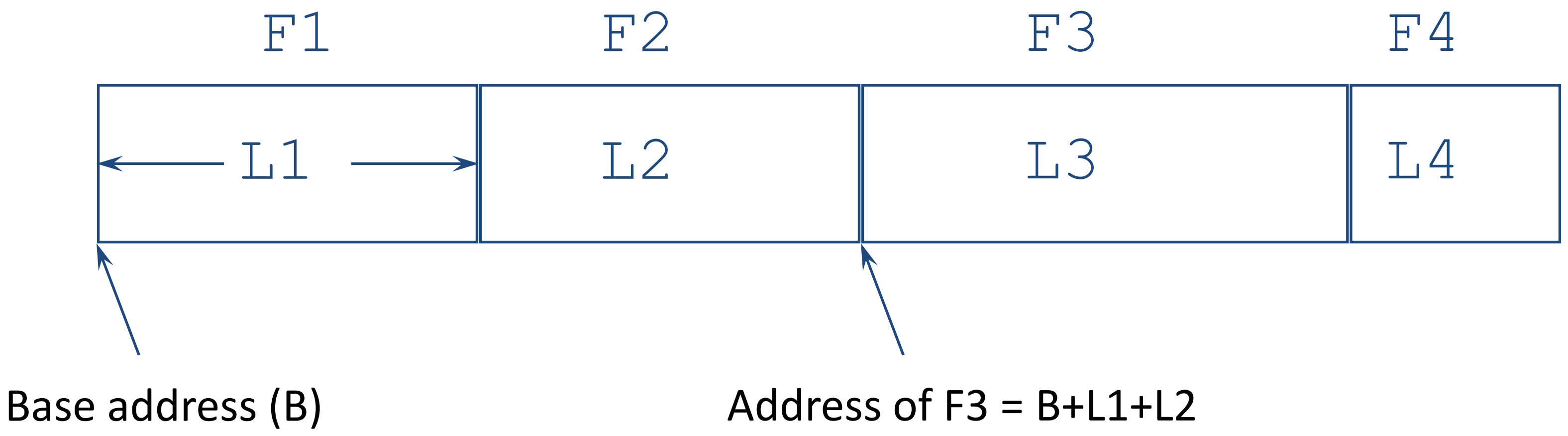


Example: consider the enrollment table *E(sid: INT4, semester: CHAR(3), cno: INT4, grade: FLOAT)*



Record format: fixed-length

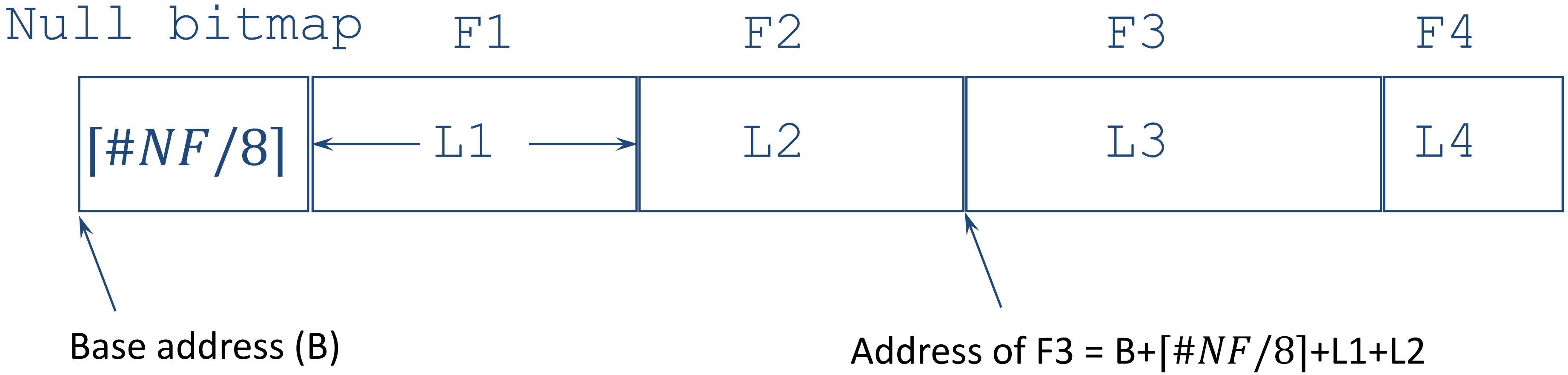
- Fixed-length record
  - How to handle NULLs?



Record format: fixed-length

- Fixed-length record
  - How to handle NULLs?
    - *Null bitmap*: set the $\mathrm{i^{th}}$ bit if Fi is NULL. Otherwise, clear the $\mathrm{i^{th}}$ bit.

#NF: number of nullable fields



Example: consider the enrollment table *E(sid: INT4, semester: CHAR(3), cno: INT4, grade: FLOAT), NF = 4*

Address alignment in records

- Address alignment requirements?
  - Alignment example: to read/write a 32-bit integer *in memory*, $\textit{its address mod } 4 == 0$
  - Most architecture has address alignment requirements
    - Some strictly enforces 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

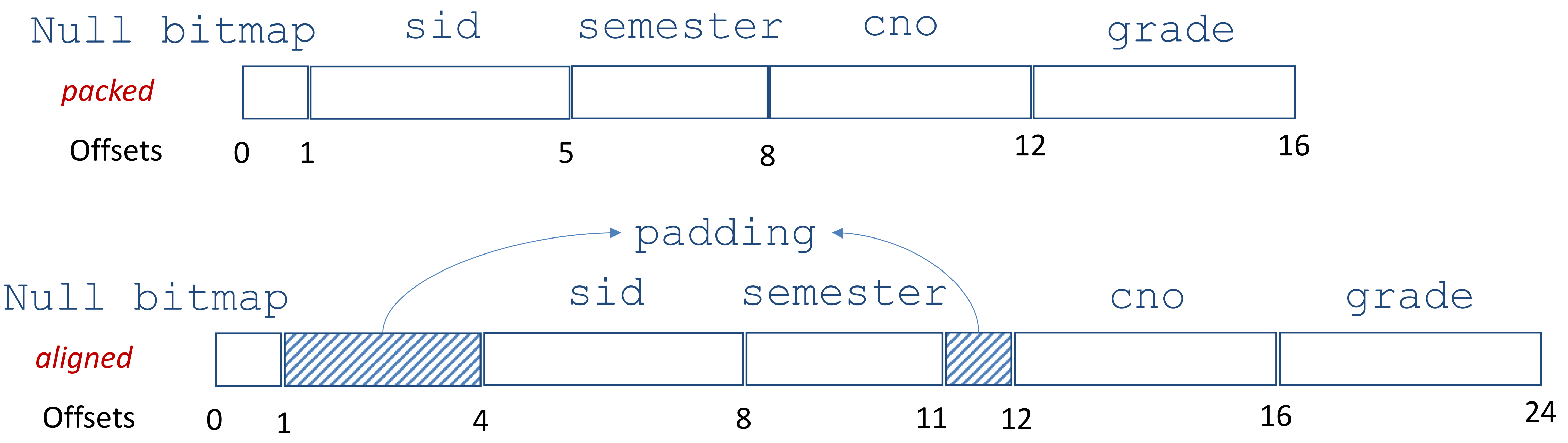
  - DB records? Two choices:
    - Pack everything, and memcpy the field before access
      - Less efficient, but save space
    - Align offsets manually
      - More efficient field access, but waste space

```
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!)
```

Address alignment in records

- Example: consider the enrollment table E(sid: INT4, semester: CHAR(3), cno: INT4, grade: FLOAT), NF = 4
  - alignment requirements
    - INT4: 4
    - CHAR: 1
    - FLOAT: 4



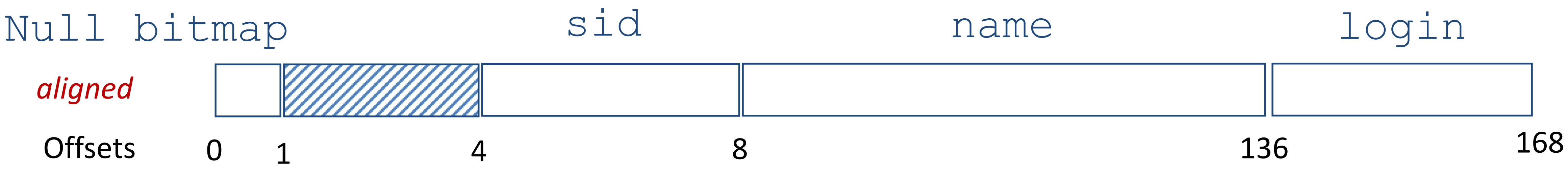
Is there any assumption for the fields in an aligned record to be really aligned?

Base address B must be aligned to the strictest alignment requirement. (depends on arch, OS and DB type system)

Record format: fixed-length

- Problem with fixed-length record?
  - What if we have a variable-length field whose maximum length >> average length
    - Wastes space

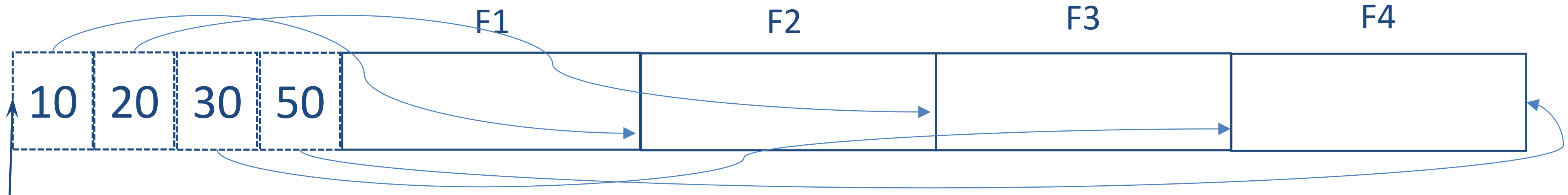
Example: consider the student table *S(sid: INT4, name: VARCHAR(128), login: VARCHAR(32))*



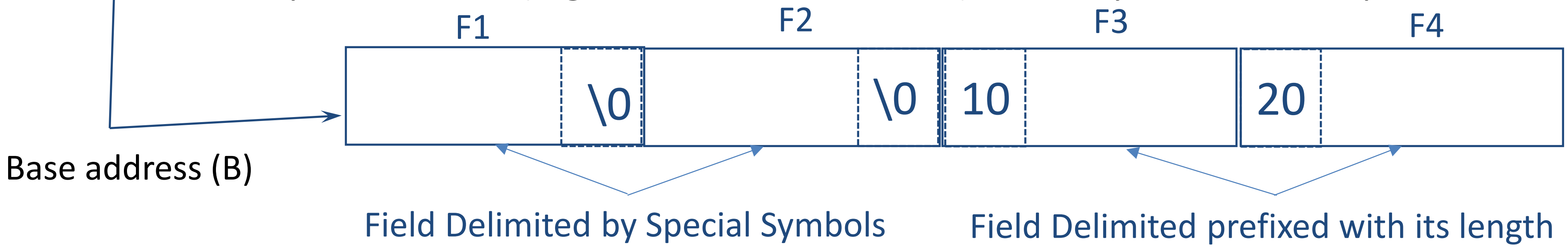
Solution: variable-length records

Record format: variable-length

- Variable-length record
  - Two approaches:
    - Encode field length in an offset array (e.g., stores the end offset of each field)
      - random access to fields given B, but takes more space

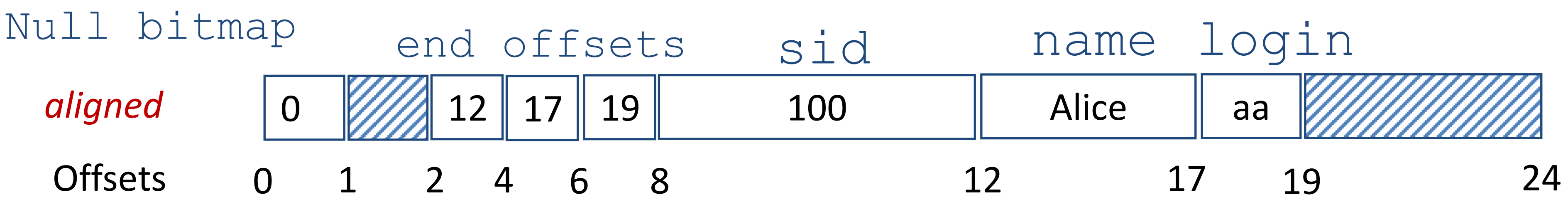


    - Using self-contained data field (with separator/encoded length)
      - Computed offsets (e.g., offset of F3 = L1 + L2); but may be more compact

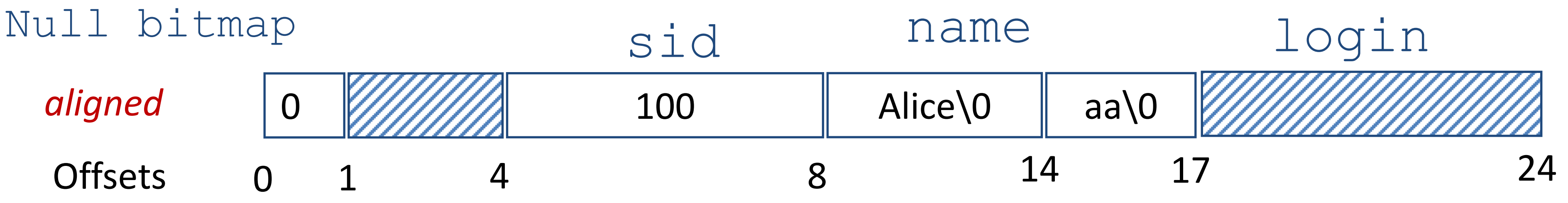


Record format: variable-length

- Example: consider a record in S with (sid = 100, name = ‘Alice’, login = ‘aa’), NF = 3
  - Two approaches:
    - Encode field length in an offset array (e.g., stores the end offset of each field)
      - assuming offsets are stored as `int16_t`

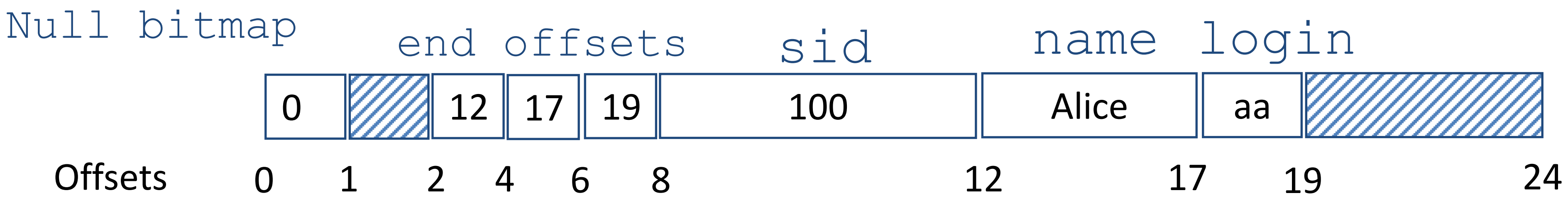


    - Using self-contained data field (with separator/encoded length)
      - Computed offsets (e.g., offset of F3 = L1 + L2); but may be more compact



Record format: variable-length

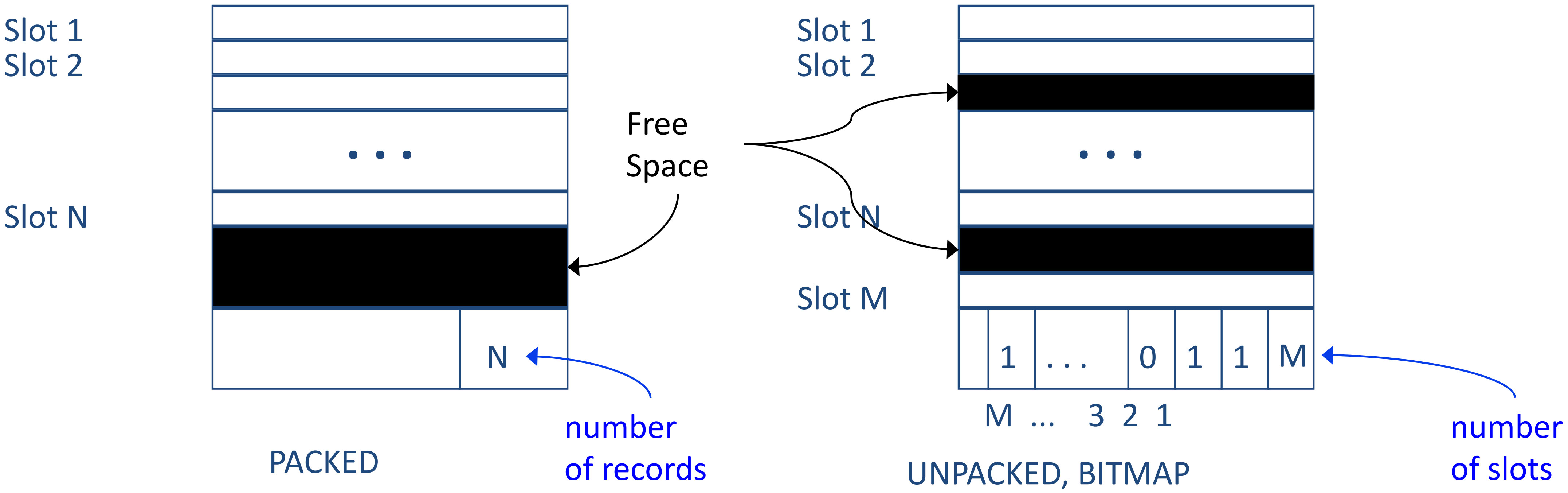
- Example: consider a record in S with (sid = 100, name = ‘Alice’, login = ‘aa’), NF = 3
  - Many possible designs with minor tweaks for different space/time efficiency trade-offs
    - Can also combine both fixed-length and variable-length record formats
  - Encode field length in an offset array (e.g., stores the end offset of each field)
    - assuming offsets are stored as `int16_t`



- *Example tweaks and assumption:*
  - *Fixed-length fields appear before variable-length fields => have fixed offsets*
  - *(Real) record length without the trailing padding stored somewhere else*

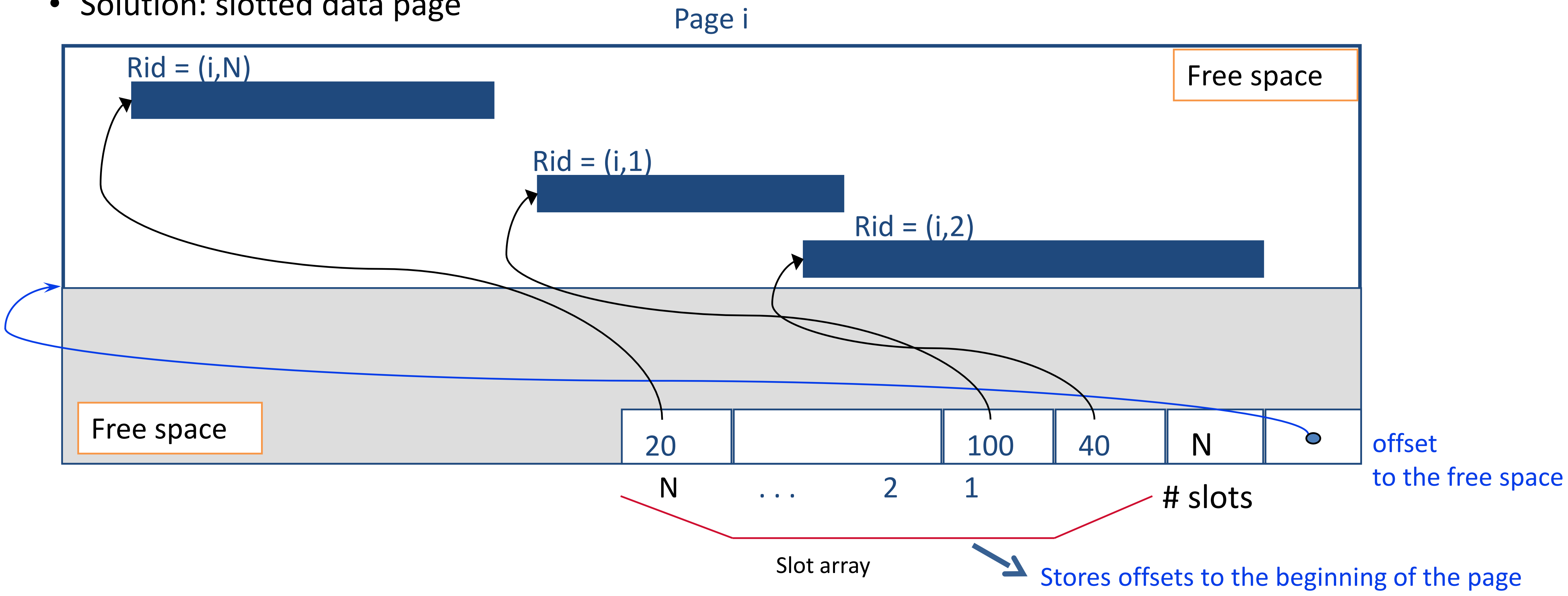
Page layout for fixed-length records

- Why not storing record consecutively in a file?
  - Linear time to update/delete!
- How do we store records in fixed-size pages?
  - Fixed-length record: easy (packed vs unpacked)
    - Not commonly used as it wastes space



Page layout for variable-length records

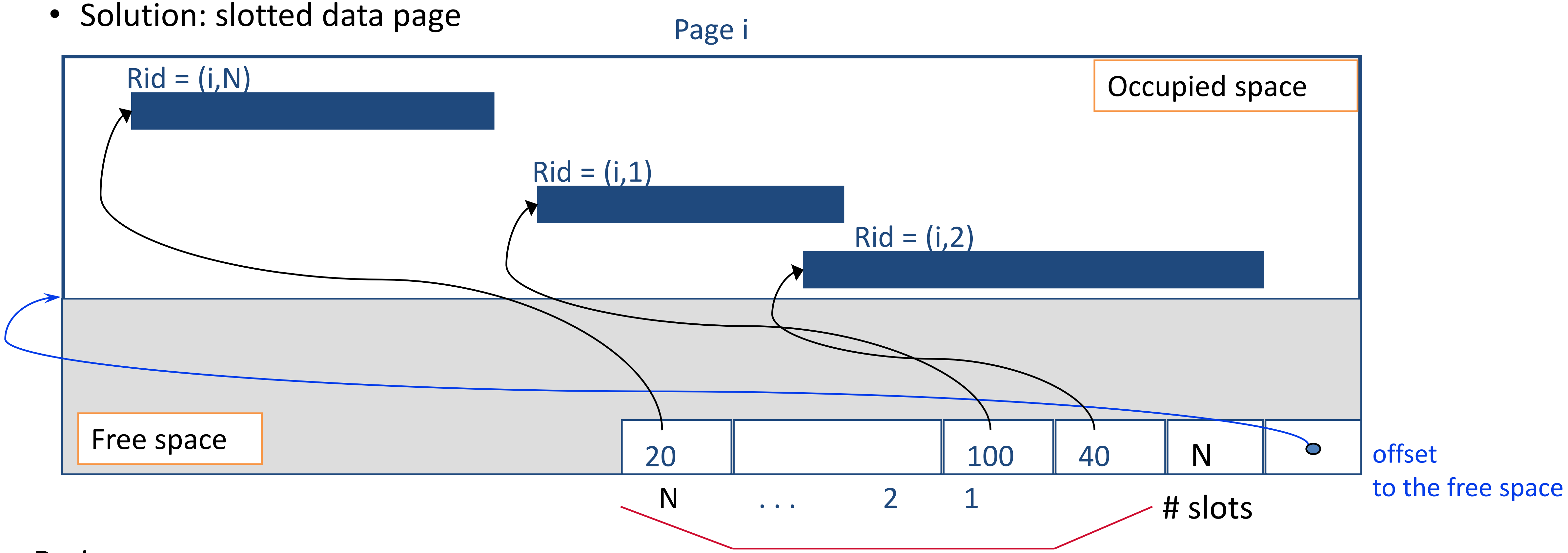
- What about variable-length records?
  - Solution: slotted data page



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

Page layout for variable-length records

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



Design space:

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

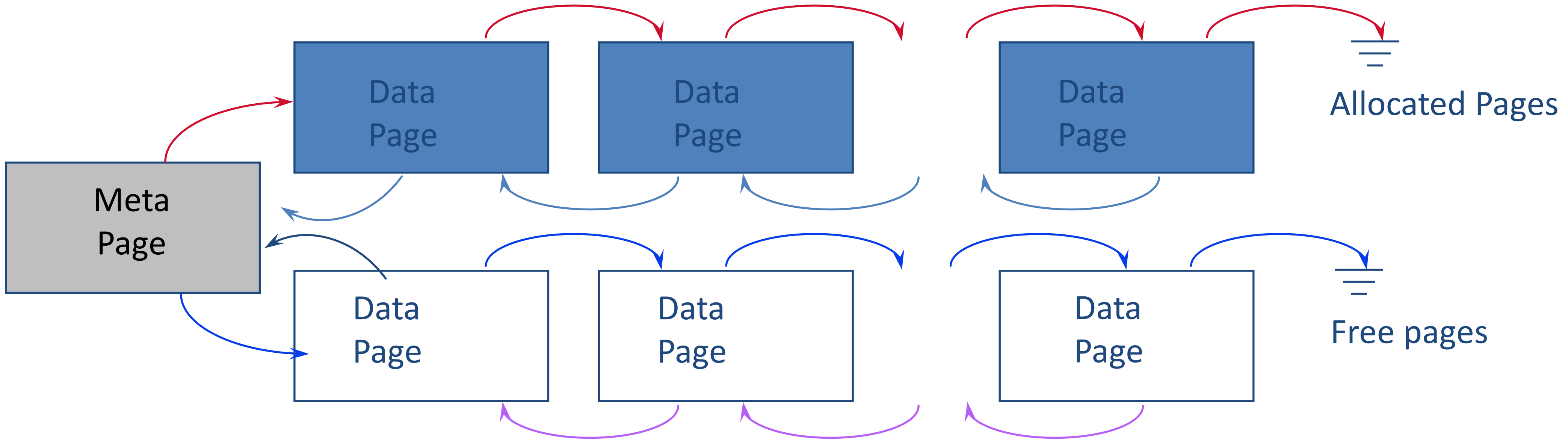
Organizing pages in a heap file

- 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

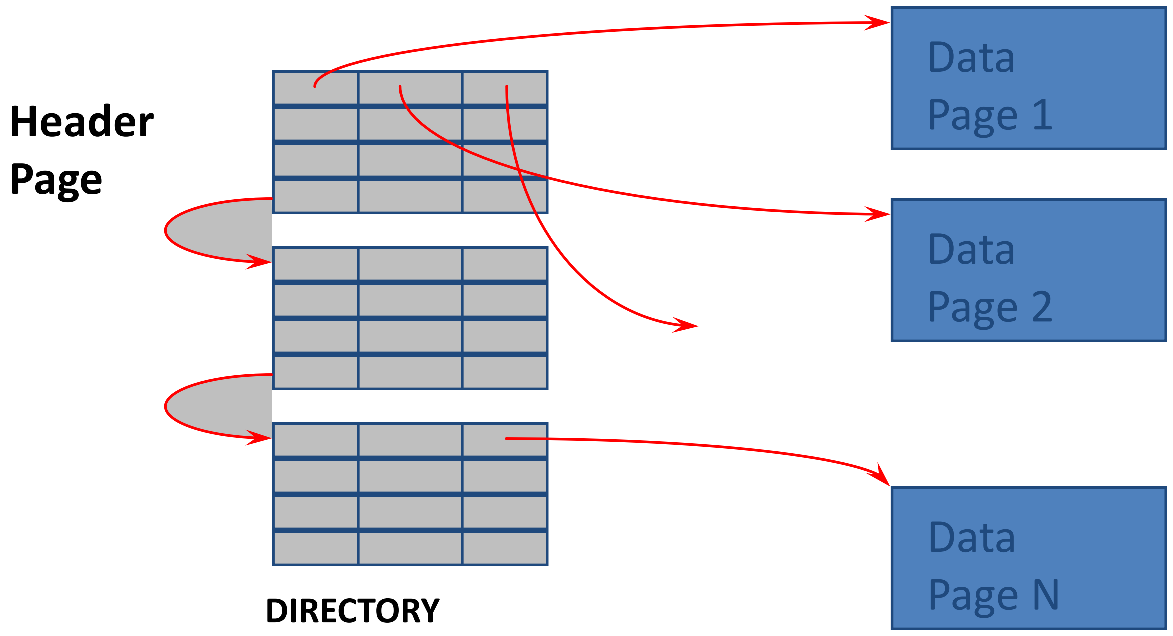
- Many possible alternatives and variants
  - We consider the most representative two of them

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 in a (separate) 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*!

Database catalog

- How does DBMS remember the layout?
- 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