Discussion 2

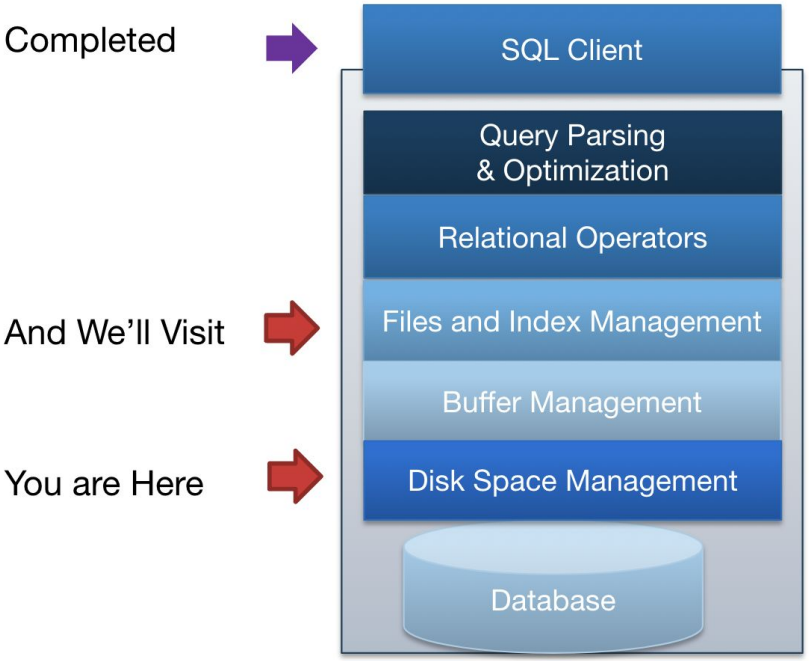
Disks, Files, Buffers

Announcements

Project 1 due **Fri. 2/2** at 11:59PM PST!

Vitamin 2 due **Mon. 2/5** at 11:59PM PST.

Recap / Discussion Motivation



Last time, we introduced SQL, which is primarily used to query databases.

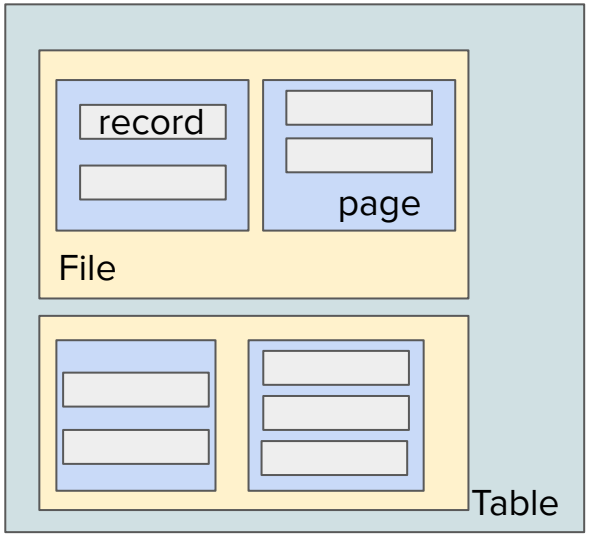
This week, we'll begin breaking down how databases and tables are organized and how they gather data you requested in your queries.

Page/Record Formats within Databases

Overview

- Tables are stored as logical files consisting of pages
- Each page contains a collection of records

- Table (many per database)
  - File (corresponds to a table)
    - Page (many per file)
      - Record (many per page)



Overview: Pages and I/Os

- Pages are managed
  - in memory by the buffer manager: higher levels of database only operate in memory
  - on disk by the disk space manager: reads and writes pages to physical disk/files
- Unit of accesses to physical disk is the page
  - **Cannot** fetch fractions of a page
  - I/O: unit of transferring a page of data between memory and disk (read OR write 1 page = 1 I/O)

Records

How do we organize fields within a record?

Data Types

Fixed or Variable?	Data Type	Size
Fixed	integer	4 B
Fixed	float	4 B
Fixed	boolean	1 B
Fixed	char(20)	20 B
Fixed	byte	1 B
Variable	text	>= 0 B
Variable	varchar(20)	<= 20 B

Fixed Length Records

Records are made up of multiple **fields** (think: values for columns in a table).

We have **fixed length records** when field lengths are consistent: the first field always has N bytes, the second field always has M bytes, etc.

Thus, record lengths are fixed: every record is always the same number of bytes.

For example:

```
Students (
    sid integer
    PRIMARY KEY,
    enrolled boolean,
    grad_yr integer,
);
```

	sid	enrolled	grad_yr	Size
Record1 =	1234	True	2023	9B
Record2 =	5678	False	2025	9B
Record3 =	9012	True	2024	9B

Variable Length Records

We have **variable length records** when we don't satisfy the conditions for fixed length records, that is:

- Field lengths are not consistent: eg. the third field may take 0 to 4 bytes
- Record lengths are not fixed: records can take different number of bytes

For example:

```
Students (
    sid integer
    PRIMARY KEY,
    name varchar,
    grad_yr integer,
);
```

	sid	name	grad_yr	Size
Record1 =	1234	Ben	2023	11B
Record2 =	5678	Rithvik	2025	15B
Record3 =	9012	Kayla	2024	13B

Variable Length Records

Two ways to store variable length records:

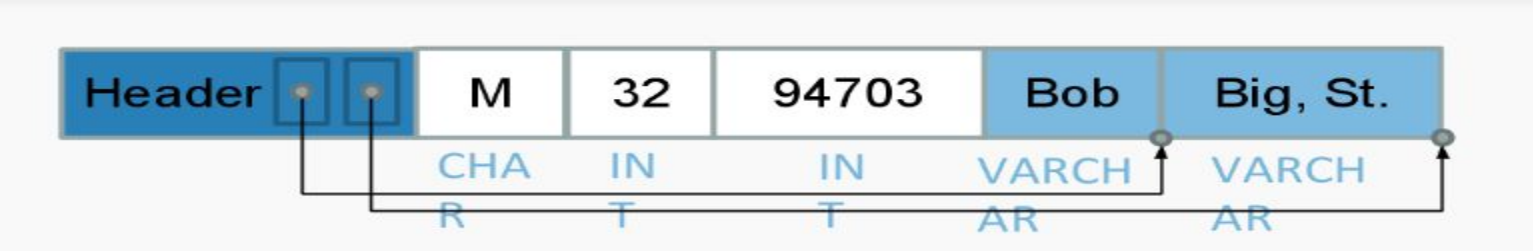
- Delimit fields with a special character ($ in the diagram below)

  - What if F2 contains ‘$’?

Variable Length Records

Two ways to store variable length records:

- Array of field offsets

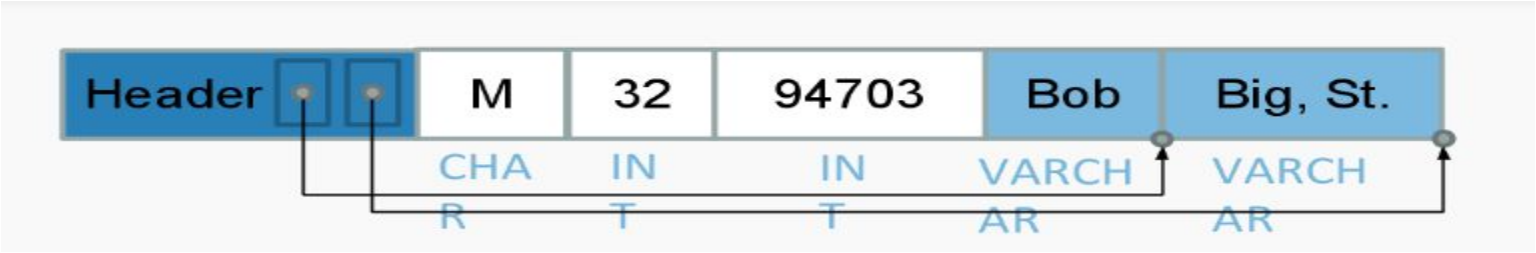


  - Each record contains a **record header**
  - Variable length fields are placed *after* fixed length fields
  - Record header stores **field offset** indicating where each variable length field ends
    - Why do we point to the end of the field instead of the beginning?

Variable Length Records

Two ways to store variable length records:

- Array of field offsets



  - An aside: this is not actually sufficient for storing NULLs
    - Cannot distinguish between empty string (“”) and NULL
    - Need some extra metadata (e.g. bitmap in record header or special char in field), which varies widely between different DBMS

Pages

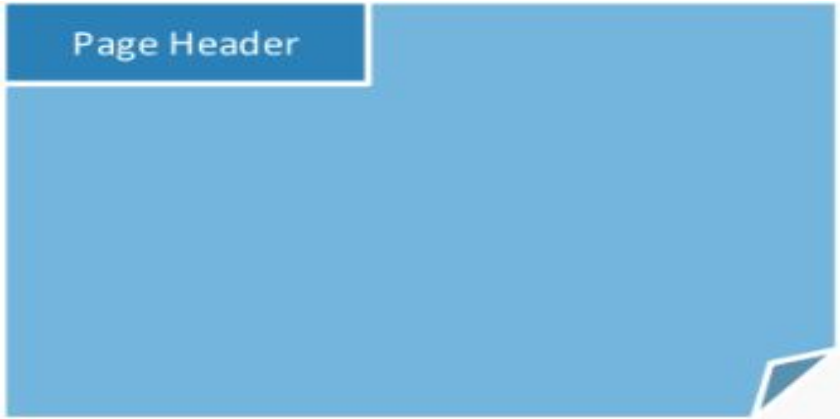
How do we organize records within a page?

Page basics: the header

The **page header** is a portion of each page reserved to keep track of the records in the page.

The page header may contain fields such as:

- Number of records in the page
- Pointer to segment of free space in the page
- Bitmap indicating which parts of the page are in use

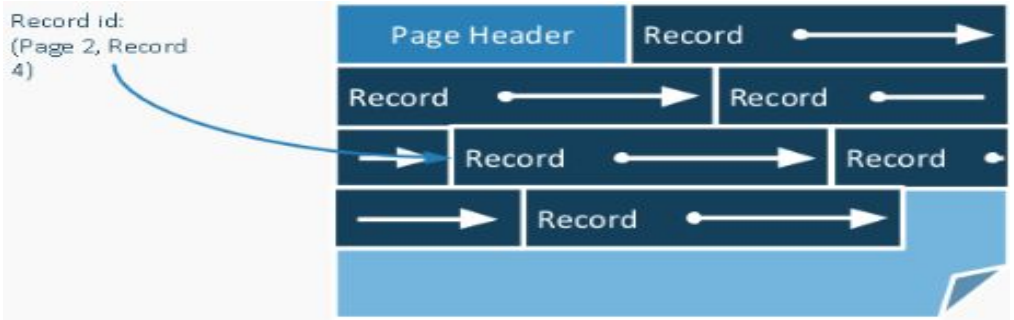


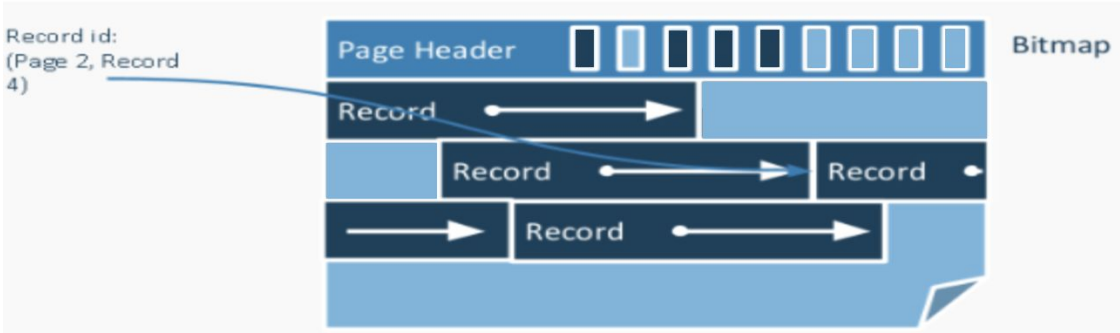
Fixed Length Records

We can store fixed length records in two ways:

- **packed**: no gaps between records, record ID is location in page

- **unpacked**: allow gaps between records, use a bitmap to keep track of where the gaps are



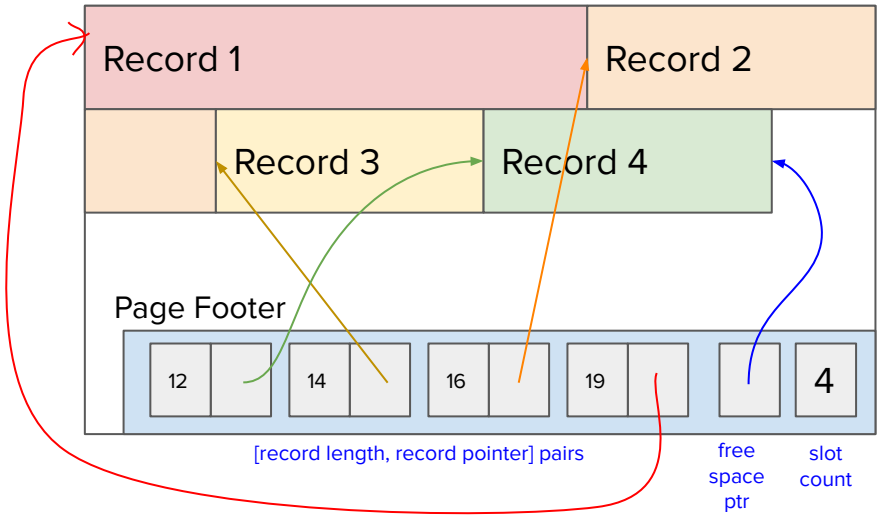


Variable Length Records

- How do we know where each record begins?
- What happens when we add and delete records?

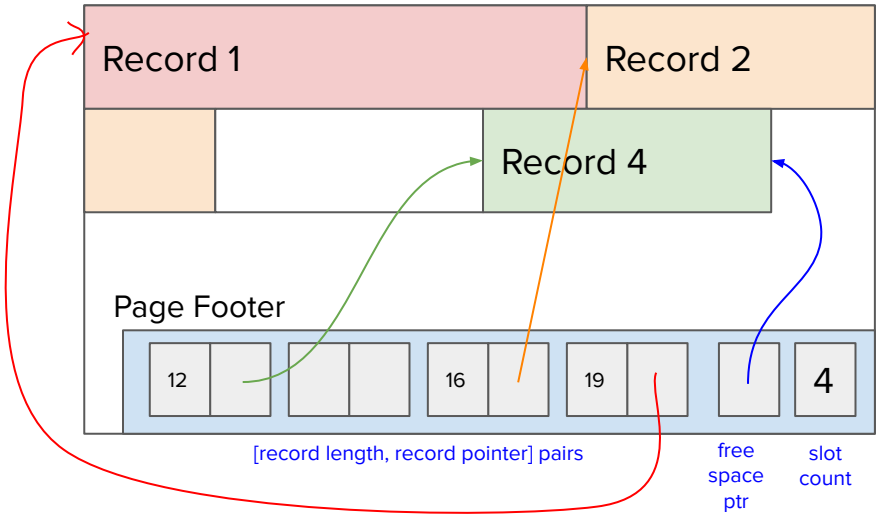
Variable Length Records: Slotted Pages

- Move page header to *end* of page (footer) - to allow for “header” to grow
- Store length and pointer to start of each record in footer
- Store number of slots and pointer to free space

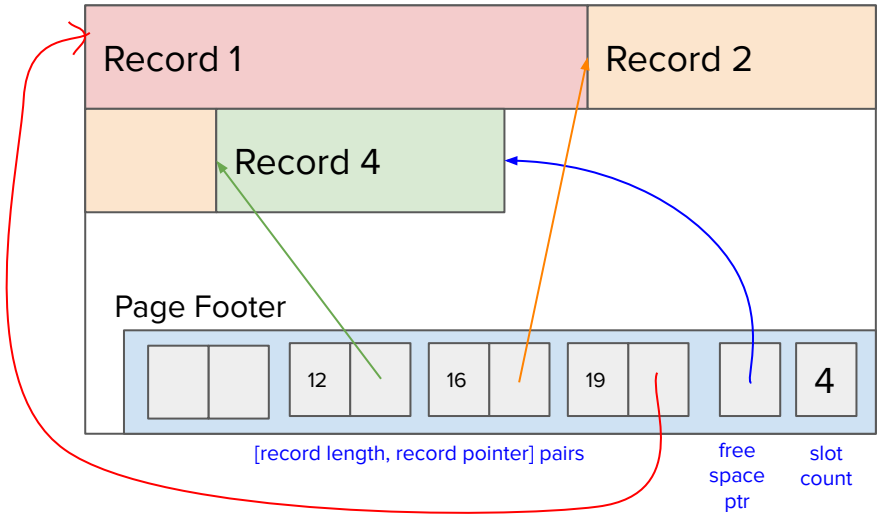


Slotted Page: Packed vs. Unpacked Layout

Delete Record 3:



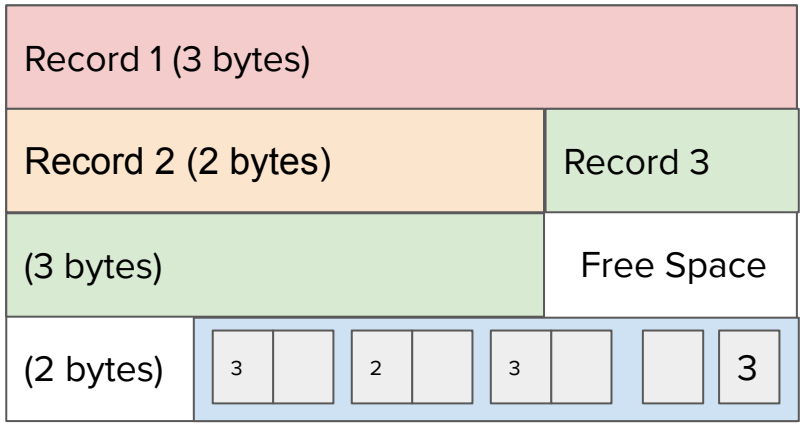
Slotted Page (Unpacked Layout)



Slotted Page (Packed Layout)

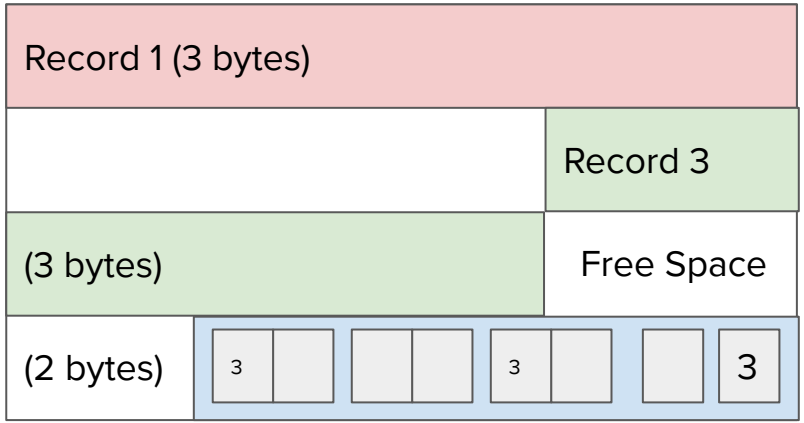
Variable Length Records: Fragmentation

- Deleting records causes fragmentation if we use an **unpacked** layout:
  - Say we have the following data page
  - 3 byte record, 2 byte record, 3 byte record, 2 byte free space
  - If we delete the 2 byte record, we have 4 bytes free on the page but cannot insert a 4 byte record



Variable Length Records: Fragmentation

- Deleting records causes fragmentation if we use an **unpacked** layout:
  - Say we have the following data page
  - 3 byte record, 2 byte record, 3 byte record, 2 byte free space
  - If we delete the 2 byte record, we have 4 bytes free on the page but cannot insert a 4 byte record



File Organization

Heap Files and Sorted Files

A **heap file** is a file with no order enforced.

- Within a heap file, we keep track of pages
  - Within a page, keep track of records (and free space)
    - Records placed arbitrarily across pages

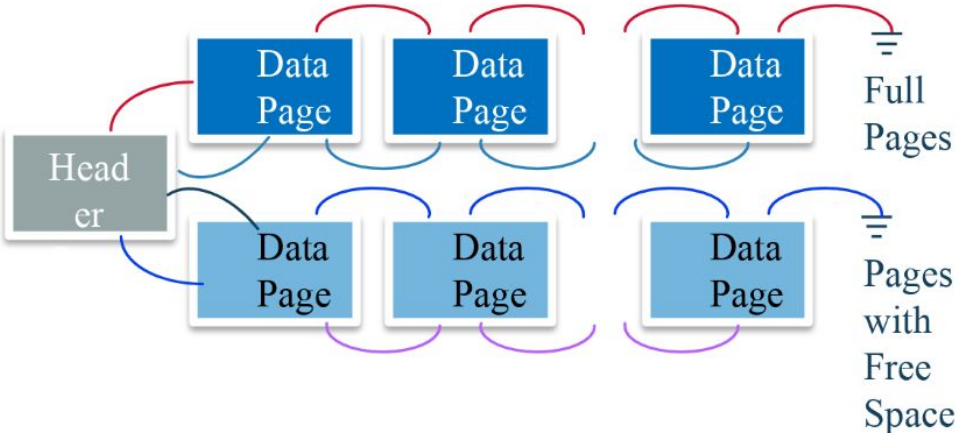
Record ID (RID) = <page id, slot #>

A **sorted file** is similar to a heap file, except we require it be sorted on a key (a subset of the fields).

Implementing Heap Files

One approach to implementing a heap file is as a **linked list**.

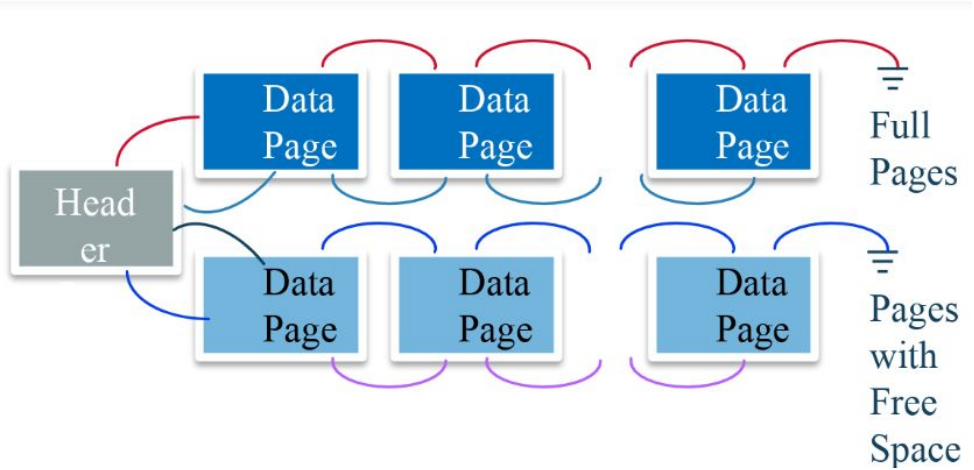
- Each page has two pointers, free space, and data.
- We have two linked lists of pages, both connected to a **header page**
  - List of full pages
  - List of pages with some empty space



Implementing Heap Files

One approach to implementing a heap file is as a **list**.

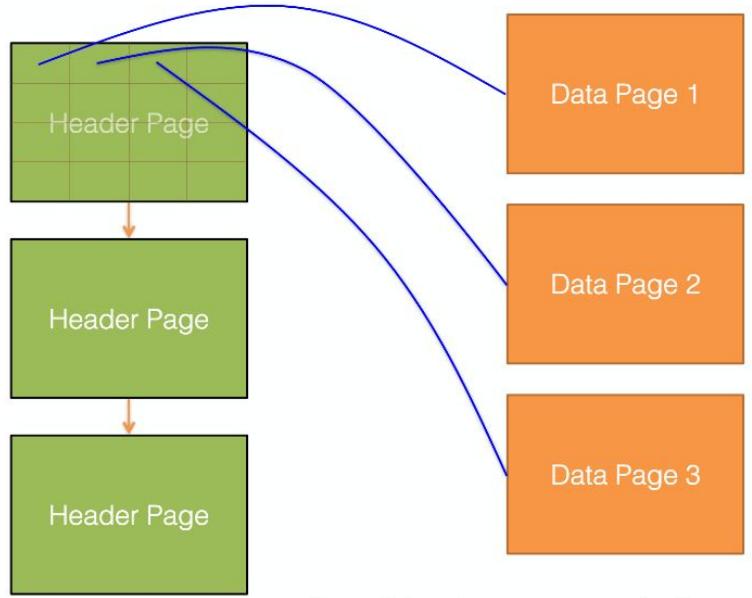
- How do we find a page to insert a 20-byte record in?



Implementing Heap Files

A different approach to implementing a heap file is with a **page directory**.

- We have a linked list of **header pages**, which are each responsible for a set of data pages
- Each entry in a header page contains:
  - Pointer to a data page
  - The amount of free space for the data page



Worksheet

True and False - A

When querying for an 16 byte record, exactly 16 bytes of data is read from disk.

True and False - A

When querying for an 16 byte record, exactly 16 bytes of data is read from disk.

False, an entire page of data is read from disk.

True and False - B

In a heap file, all pages must be filled to capacity except the last page.

True and False - B

In a heap file, all pages must be filled to capacity except the last page.

False, there is no such requirement.

True and False - C

Assuming integers take 4 bytes and pointers take 4 bytes, a slot directory that is 512 bytes can address 64 records in a page.

True and False - C

Assuming integers take 4 bytes and pointers take 4 bytes, a slot directory that is 512 bytes can address 64 records in a page.

False, we have the free space pointer, which doesn't fit after 64 * (4 + 4) = 512 bytes of per-record data in the slot directory.

True and False - D

In a page containing fixed-length records with no nullable fields, the size of the bitmap never changes.

True and False - D

In a page containing fixed-length records with no nullable fields, the size of the bitmap never changes.

True, the size of the records is fixed, so the number we can fit on a page is fixed.

Fragmentation and Record Formats #1

Is fragmentation an issue with fixed length records on an unpacked page?

Fragmentation and Record Formats #1

Is fragmentation an issue with fixed length records on an unpacked page?

No. Since all records are the same size (fixed), we'll never encounter a case where we can't add a record to a page even though the record's size is less than or equal to the total amount of free space on the page.

Fragmentation and Record Formats #2

Is fragmentation an issue with variable length records on an unpacked slotted page?

Fragmentation and Record Formats #2

Is fragmentation an issue with variable length records on an unpacked slotted page?

Yes. For example, even if a page has n bytes of space dispersed throughout, we may not be able to add a n-byte record because the space is not consolidated together. This can happen because variable length records can be of any size, causing gaps of varying sizes upon deleting them.

Record Formats (a)

Assume we have a table that looks like this:

```
CREATE TABLE Questions (
    qid integer PRIMARY KEY,
    answer integer,
    qtext text,
);
```

Recall that integers and pointers are 4 bytes long. Assume for this question that the record header stores pointers to all of the variable length fields (but that is all that is in the record header). How many bytes will the smallest possible record be?

Record Formats (a)

Assume we have a table that looks like this:

```
CREATE TABLE Questions (
    qid integer PRIMARY KEY,
    answer integer,
    qtext text,
);
```

Recall that integers and pointers are 4 bytes long. Assume for this question that the record header stores pointers to all of the variable length fields (but that is all that is in the record header). How many bytes will the smallest possible record be?

12 bytes. The record header will only contain one pointer to the end of the qtext field so it will only be 4 bytes long. The qid and answer fields are both integers so they are 4 bytes long, and in the smallest case qtext will be 0 bytes. This gives us a total of 12 bytes.

Record Formats (b)

Assume we have a page that is 1 KiB in size. What is the maximum number of records that can fit on a page?

Record Formats (b)

Assume we have a page that is 1 KiB in size. What is the maximum number of records that can fit on a page?

50 records. If we have 1 KiB for data, this means we have 1024 bytes to use for all the records and the page footer slot directory. The slot directory always stores the slot count (4 bytes) and a free space pointer (4 bytes). Thus, 1024 bytes - 4 bytes (slot count) - 4 bytes (free space pointer) = 1016 bytes remaining for records and corresponding slots. In addition, for each record we store on the page, it has a corresponding slot in the page footer that takes up 8 bytes (4 byte record pointer and 4 byte record length). The minimum record size (12 bytes) and each slot takes 8 bytes, so we divide 1016 by 20. Since we can't have fractional records on a page, we round down to the nearest record. This is equal to floor(1016 / 20) = 50.

Record Formats (c)

Now assume the fields can be null so we add a bitmap to the beginning of our record header indicating whether or not each field is null. Assume this bitmap is padded so that it takes up a whole number of bytes (i.e. if the bitmap is 10 bits it will take up 2 full bytes). How big is the largest possible record assuming that the qtext is null?

Record Formats (c)

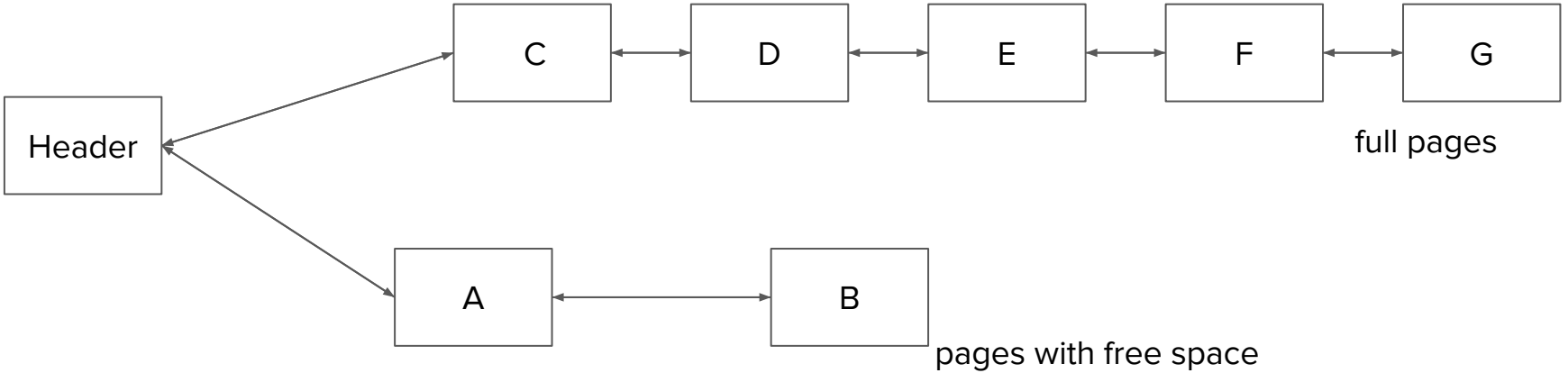
Now assume the fields can be null so we add a bitmap to the beginning of our record header indicating whether or not each field is null. Assume this bitmap is padded so that it takes up a whole number of bytes (i.e. if the bitmap is 10 bits it will take up 2 full bytes). How big is the largest possible record assuming that the qtext is null?

13 bytes. The qid cannot be null because it is a primary key so there will be 2 slots in our bitmap (and thus 2 bits), meaning our record header will only be 1 byte longer than it was in part a. In the max case, both the qid and answer field will be present and still be 4 bytes each. Therefore, the fields haven't changed at all, and the total record length is now 13 bytes.

Calculate the I/Os (Linked List Implementation)

Assume we have a heap file implemented with a linked list. The heap file contains 5 full pages and 2 pages with free space, at least one of which has enough space to fit the record we are trying to insert.

(a) In the worst case, how many I/Os are required to find a page with enough free space to insert a record?


Header
C
D
E
F
G
full pages
A
B
pages with free space

Calculate the I/Os (Linked List Implementation)

Assume we have a heap file implemented with a linked list. The heap file contains 5 full pages and 2 pages with free space, at least one of which has enough space to fit the record we are trying to insert.

(a) In the worst case, how many I/Os are required to find a page with enough free space to insert a record?

3 I/Os:

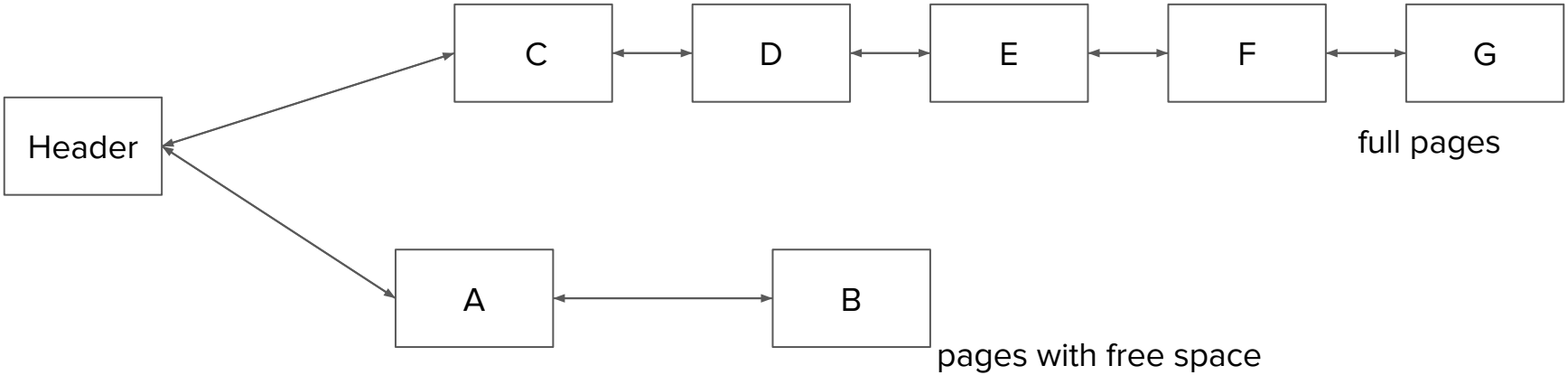
- We incur 1 I/O to read in the header page.
- In the worst case, only the final page in the list of pages with free space has enough space for this record. Thus, we read in the 2 pages in the list of pages with free space, incurring 2 I/Os.

Calculate the I/Os (Linked List Implementation)

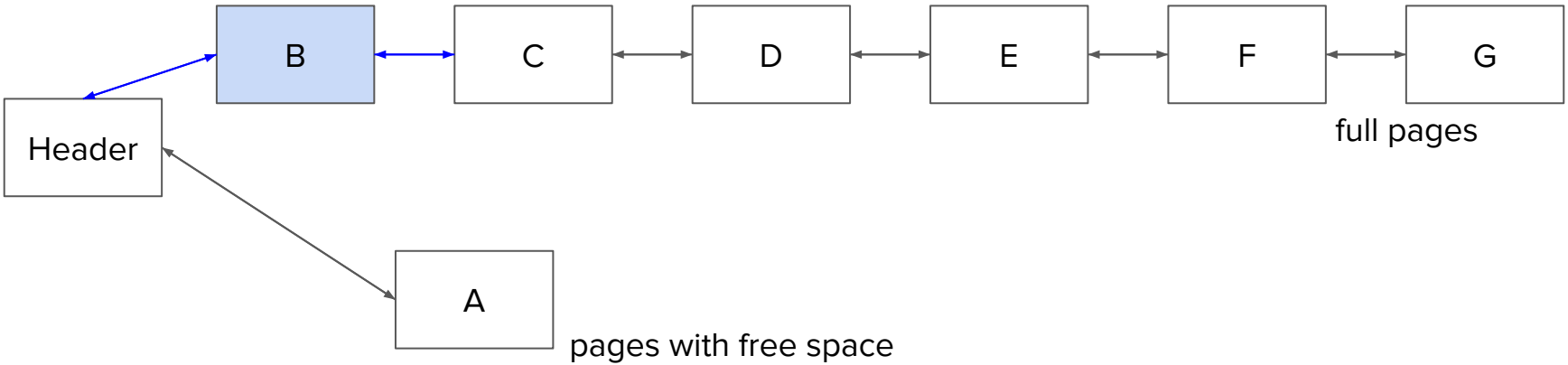
Assume we have a heap file implemented with a linked list. The heap file contains 5 full pages and 2 pages with free space, at least one of which has enough space to fit the record we are trying to insert.

(b) In the worst case, how many I/Os are required to insert a record into the 2nd page with free space? Consider what happens when after inserting the record, this page becomes full. Also, assume that we can modify the header page and insert to the beginning of the full pages linked list.

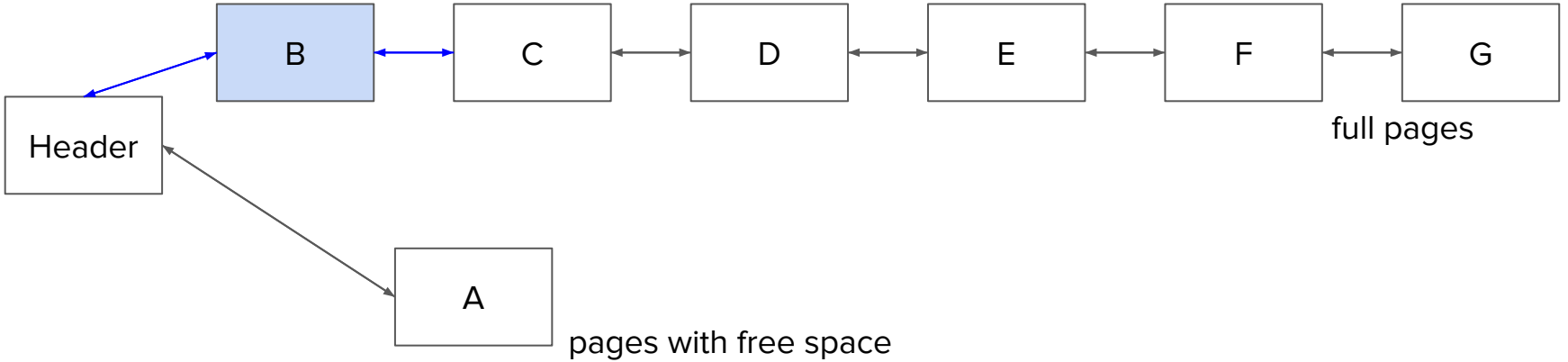
Before



After



After



Total: 8 I/Os

- Read header page (1 I/O) to get where the first free and full pages are
- Read page A (1 I/O) to get where page B is
- Read page B (1 I/O) to:
  - Add the record
  - Change its left pointer to the header page
  - Change its right pointer to point to page C
- Write page B to disk (1 I/O)
- Write page A (1 I/O) because we need to change page A's right pointer to no longer refer to page B,
- Write header page (1 I/O): need to change header page's pointer for list of full pages to refer to page B
- Read page C (1 I/O)
- Write page C (1 I/O): need to change left pointer to refer to page B

Calculate the I/Os (Page Directory Implementation)

Assume we have a heap file implemented with a page directory containing 54 data pages. Each page in the directory can hold 16 page entries.

(a) In the worst case, how many I/Os are required to find a page with free space?

Calculate the I/Os (Page Directory Implementation)

Assume we have a heap file implemented with a page directory containing 54 data pages. Each page in the directory can hold 16 page entries.

(a) In the worst case, how many I/Os are required to find a page with free space?

4 I/Os. The page directory will contain ceil(54/16) = 4 header pages. In the worst case, we will need to look through all 4 pages in the directory, incurring a total cost of 4 I/Os.

Note that the page directory contains the amount of free space for each page, so we do not need to read in the data pages to check whether they contain free space.

Calculate the I/Os (Page Directory Implementation)

Assume we have a heap file implemented with a page directory containing 54 data pages. Each page in the directory can hold 16 page entries.

(b) In the worst case, how many I/Os are required to insert a record into this file? Assume at least one data page has enough space to fit this record.

Calculate the I/Os (Page Directory Implementation)

Assume we have a heap file implemented with a page directory containing 54 data pages. Each page in the directory can hold 16 page entries.

(b) In the worst case, how many I/Os are required to insert a record into this file? Assume at least one data page has enough space to fit this record.

7 I/Os. From the previous part, we need 4 I/Os to find a data page with enough free space to fit this record (in the worst case). Then, we need to read this page with enough free space into memory (1 I/O), insert the record, and write this page back to disk (1 I/O). Lastly, we need to update the amount of free space for this data page in our page directory, and write the page in the directory back to disk (1 I/O).

Attendance Link

https://cs186berkeley.net/attendance